COMPUTARIUM
Lycée classique de Diekirch
32 av. de la gare L-9233 Diekirch

Microcontroller Programming Project

computarium.lcd.lu presents

FAT32 on an SD Card

Reading and Writing Files from a PIC18F452, Programmed in MyBug

A running report, built step by step

Author: LX1BW
claude.baumann@education.lu

This project has primarily educational objectives
© 2026 Claude Baumann. Licensed under CC BY-NC-SA 4.0
Version 1.0 — October 16, 2026

Abstract

This report shows, step by step, how a PIC18F452 microcontroller reads and writes files on an SD card formatted with the FAT32 file system — without any driver or file-system library. The firmware is written in MyBug, the author's transparent cross-compiler, so that every byte exchanged with the card is visible in the source. Following a constructivist method, the work is divided into nine building blocks, each adding exactly one capability and each verified on the hardware by its own test program: the SPI bus, the card initialization, reading a sector, the partition table and the boot sector, the root directory, reading a file across its cluster chain, writing a sector, appending to a file with allocation of new clusters, and creating a new file. Introductory chapters explain the SD card in SPI mode and the structure of FAT32 for students meeting both subjects for the first time, and a layer model shows how the programs divide the work from single bytes up to files. All results were cross-checked by hand calculation and on a PC, where Windows' `chkdsk` found the card free of errors after the PIC had modified and created files. Along the way, the MyBug back end for the PIC18F452 gained SPI support and its 32-bit arithmetic, 256-byte arrays and multi-parameter functions were proven on the hardware.

Contents

1 Introduction **4**
- 1.1 Goal . . . 4
- 1.2 Why the PIC18F452? . . . 4
- 1.3 Method: building blocks . . . 4
- 1.4 Design decisions . . . 4
- 1.5 How to read this report . . . 4
- 1.6 Prior work . . . 5

2 The SD card **5**
- 2.1 What is inside . . . 5
- 2.2 Two bus modes . . . 5
- 2.3 Commands and responses . . . 6
- 2.4 Reading and writing a block . . . 6
- 2.5 Initialization . . . 6
- 2.6 Card types and addressing . . . 7

3 The FAT32 file system **7**
- 3.1 Why a file system? . . . 7
- 3.2 From the whole card to a single file . . . 7
- 3.3 The FAT: a linked list in a table . . . 8
- 3.4 Directories and directory entries . . . 8
- 3.5 What the programs leave out . . . 8

4 From byte to file: the layers of the programs **8**

5 Hardware **9**
- 5.1 The SPI connection . . . 9
- 5.2 Level shifting . . . 9
- 5.3 The AZ-Delivery micro-SD module . . . 9

6 Building block B1: the SPI bus **10**
- 6.1 What this block adds . . . 10
- 6.2 The test: a loopback without a card . . . 11
- 6.3 Program walkthrough . . . 11
- 6.4 Verification . . . 12

7 Building block B2: card initialization **12**
- 7.1 The command frame . . . 13
- 7.2 The initialization sequence . . . 13
- 7.3 Program walkthrough . . . 14
- 7.4 Expected output . . . 15
- 7.5 Measuring the MISO level . . . 15
- 7.6 Verification . . . 15

8 Building block B3: reading one sector **15**
8.1 What this block adds . . . 15
8.2 The data packet . . . 16
8.3 Two halves of 256 bytes . . . 16
8.4 Program walkthrough . . . 16
8.5 Expected output . . . 17
8.6 Verification . . . 17

9 Building block B4: partition table and boot sector **17**
9.1 The layout of a FAT32 partition . . . 17
9.2 What this block adds . . . 18
9.3 Program walkthrough . . . 18
9.4 Expected output . . . 19
9.5 Verification . . . 20

10 Building block B5: the root directory **20**
10.1 From cluster number to sector number . . . 20
10.2 A state machine with one read . . . 20
10.3 Processing a directory sector . . . 21
10.4 Program walkthrough . . . 21
10.5 Expected output . . . 22
10.6 Verification . . . 23

11 Building block B6: reading a file **23**
11.1 What this block adds . . . 23
11.2 Checking the transfer . . . 23
11.3 Program walkthrough . . . 24
11.4 Expected output . . . 24
11.5 Verification . . . 25

12 Building block B7: writing one sector **25**
12.1 A safe test sector . . . 25
12.2 The CMD24 sequence . . . 25
12.3 Six operations, one read and one write site . . . 25
12.4 Program walkthrough . . . 26
12.5 Expected output . . . 27
12.6 Verification . . . 27

13 Building block B8: appending to a file **27**
13.1 What has to change on the card . . . 27
13.2 The state machine . . . 28
13.3 Where the end of the file is . . . 28
13.4 Allocating a cluster . . . 28
13.5 Program walkthrough . . . 28
13.6 Expected output . . . 30
13.7 Verification . . . 31

14 Building block B9: creating a file **31**
14.1 What this block adds . . . 31
14.2 Program walkthrough . . . 32
14.3 Expected output . . . 33
14.4 Verification . . . 33

15 Conclusion **34**
15.1 What has been achieved . . . 34
15.2 What there is to learn . . . 34
15.3 What MyBug gained . . . 34

16 Outlook **34**

License, trademarks and acknowledgments **35**

A Listings **36**
A.1 Test program 18_sd_1_spi.jan 36
A.2 Test program 18_sd_2_init.jan 37
A.3 Test program 18_sd_3_read.jan 39
A.4 Test program 18_sd_4_bpb.jan 43
A.5 Test program 18_sd_5_dir.jan 48
A.6 Test program 18_sd_6_read_file.jan 53
A.7 Test program 18_sd_7_write.jan 59
A.8 Test program 18_sd_8_append.jan 64
A.9 Test program 18_sd_9_create.jan 74
A.10 SPI snippets in _pic18Fxx_.jan 84
A.11 SPI pins in _specific_pic18F452.jan 85

1 Introduction

1.1 Goal

A small microcontroller that writes its measurements to an SD card produces data that any PC can read without special software — provided the card carries a standard file system. This project implements the FAT32 file system (8.3 file names only) on top of the SD card's SPI interface, on a PIC18F452 running at 16 MHz (HS crystal).

1.2 Why the PIC18F452?

A FAT32 implementation needs one complete 512 B sector in RAM. The PIC16F88 (368 bytes of RAM) cannot hold one; the PIC18F452 offers 1536 B of RAM and 32 kB of program memory, and MyBug already has a working PIC18F452 back end (outputs, UART, timer interrupts, ADC, look-up tables and multitasking verified on the hardware in October 2026).

1.3 Method: building blocks

Each building block is one test program in `User code/18_sd_n_*.jan`:

B1 SPI bus — MSSP master, loopback test without a card.

B2 Card initialization — CMD0, CMD8, ACMD41, CMD58 (SDHC detection).

B3 Reading one sector — CMD17, hex dump of sector 0 over the UART.

B4 Partition table and boot sector — locating the FAT and the data area.

B5 Root directory — listing the 8.3 entries.

B6 Reading a file — following the cluster chain.

B7 Writing one sector — CMD24, read back and compare.

B8 Appending to a file — extending the cluster chain, updating size and FATs.

B9 Creating a file — a new directory entry; a logger demonstration.

1.4 Design decisions

- **Scope:** reading *and* writing from the start; 8.3 names only (no long file names).
- **Sector buffer:** two arrays of 256 B each. MyBug currently limits PIC18F452 arrays to 256 B (issue ISS-004); instead of changing the compiler, a byte offset $o \in [0, 511]$ is split into the half $\lfloor o/256 \rfloor$ and the index $o \bmod 256$.
- **Hardware:** a ready-made micro-SD module; its level shifting is checked (Section 5).

1.5 How to read this report

The report is written for students who know the basics of a microcontroller (ports, registers, a serial interface) but have never looked inside an SD card or a file system. Sections 2 and 3 explain the two subjects from the ground up; Section 4 shows how the programs divide the work into layers. Each building block then follows the same pattern: a *term box* introduces the new idea, a schematic shows the mechanism, a *program walkthrough* explains the important lines of the test program, and the measured output proves that the step works. The full programs are in the appendix; every one of them was run on the hardware.

1.6 Prior work

The project has a history. The author's first attempt to talk to an SD card from a PIC, in a 2008 project, failed; a second series of experiments in 2023–24 is documented in a lab report (Baumann, 2024). Those experiments started from the book by Ibrahim (2010), which builds eighteen SD card projects on PIC18 microcontrollers in C, but relies on Microchip driver libraries that are no longer available (Baumann, 2024). The lab report therefore went down to the raw SPI level: a PIC16F88 sent CMD0, CMD8, CMD58 and ACMD41 and read single blocks with CMD17, while writing blocks remained unreliable. On a Raspberry Pi Pico, a 2 GB card could finally be read and written once the block number was multiplied by 512 — an observation without a clear explanation at the time (Baumann, 2024). Block B2 of the present report supplies that explanation: such cards are *standard-capacity* cards, which are addressed in bytes, not in blocks (Section 2.6).

Microchip's own approach to the subject is a complete file-system library written in C, described in application note AN1045 (Reen & Mohanswamy, 2008). Like the libraries used by Ibrahim (2010), it hides the card and the file system behind function calls. This report goes the opposite way: no library is used, every byte that travels between the PIC and the card is visible in the MyBug source, and every number that the programs compute is checked by hand.

2 The SD card

2.1 What is inside

An SD card is not just memory. It contains flash memory *and* a small controller that manages it (Figure 1). The host never addresses flash cells directly; it sends *commands* to the controller, which reads, writes, erases and maps the flash blocks internally. For the host, the card looks like a long row of 512-byte *blocks* (sectors), numbered from 0. The authoritative description of the card is the SD Association's physical layer specification, of which a free "simplified" version exists (SD Card Association, 2023).

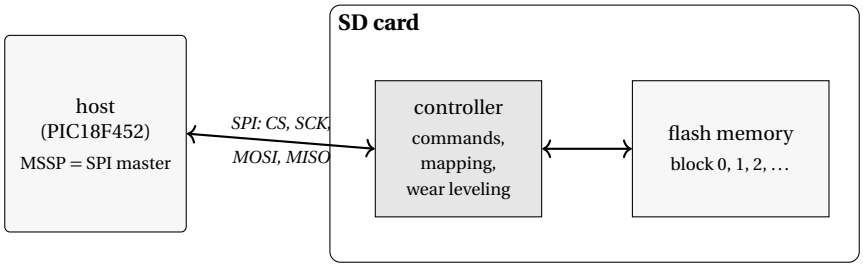


Figure 1: The host talks to the card's controller, never directly to the flash memory.

2.2 Two bus modes

A card can be operated in its native *SD mode* (a command line and up to four data lines) or in *SPI mode*, which needs only the four lines of an SPI bus. SPI mode is slower, but every small microcontroller has an SPI port, so it is the mode used here. Table 1 shows how the card's contacts are used in SPI mode.

Card contact (SD mode)	SPI mode	Direction (seen from the card)
CD/DAT3	CS (chip select, active low)	input
CMD	DI (data in = MOSI)	input
CLK	SCLK (clock from the host)	input
DAT0	DO (data out = MISO)	output
VDD, VSS	supply 2.7–3.6 V, ground	—

Table 1: SD card contacts in SPI mode (DAT1 and DAT2 are unused).

The card enters SPI mode when it receives the reset command CMD0 while CS is low; until it is powered off it then stays in SPI mode.

2.3 Commands and responses

Every command is a frame of six bytes (Figure 2): a start byte with the bit pattern `01` and the 6-bit command index, a 32-bit argument, and a byte with a 7-bit CRC and the end bit 1. In SPI mode the card checks the CRC only for CMD0 and CMD8.

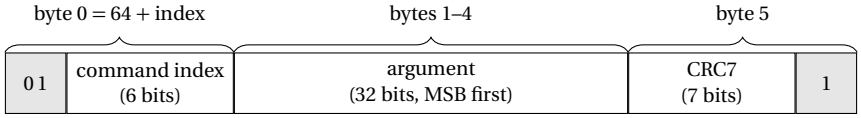


Figure 2: An SD command frame. Example CMD17 (read block 0): `51 00 00 00 00 55`.

Table 2 lists the commands used in this project. A command name starting with "A" (*application-specific command*) must be preceded by CMD55.

Command	Name	Argument	Response	Purpose
CMD0	GO_IDLE_STATE	0	R1	software reset, enter SPI mode
CMD8	SEND_IF_COND	`000001AA`	R7	check voltage range; old cards reject it
CMD17	READ_SINGLE_BLOCK	address	R1 + data	read one 512-byte block
CMD24	WRITE_BLOCK	address	R1, then data	write one 512-byte block
CMD55	APP_CMD	0	R1	"next command is an A-command"
ACMD41	SD_SEND_OP_COND	HCS bit 30	R1	start initialization; repeat until R1 = 00
CMD58	READ_OCR	0	R3	read the OCR register (CCS bit)

Table 2: The SD commands used by the test programs.

The card answers every command with the one-byte *R1* response (Figure 3); R3 and R7 are R1 followed by four more bytes. Bit 7 of R1 is always 0, which distinguishes the answer from the `FF` bytes of an idle MISO line.

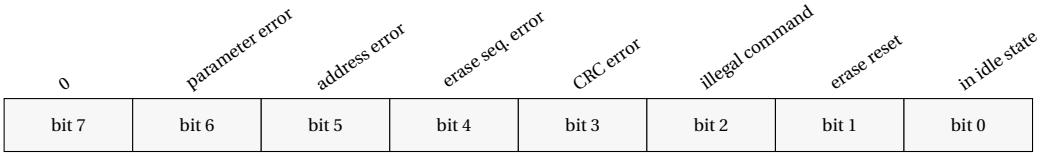


Figure 3: The R1 response byte. `01` = idle (initializing), `00` = ready, `05` = idle + illegal command (an old card rejecting CMD8).

2.4 Reading and writing a block

Data travel in *data packets*. After a read command (CMD17) and R1 = `00`, the card sends `FF` while it fetches the block, then the *start token* `FE`, the 512 data bytes and a 16-bit CRC (block B3, Figure 11). For a write (CMD24) the roles are exchanged: the host sends the token, the data and the CRC; the card answers with a *data response* (`xxx00101` = accepted) and keeps MISO low while it programs the flash (block B7, Figure 15).

2.5 Initialization

After power-up the card needs at least 74 clock cycles with CS high and a slow clock (at most 400 kHz). Then the sequence of block B2 brings it into SPI mode and through its internal initialization: CMD0, CMD8, CMD55 + ACMD41 (repeated until the card is ready) and CMD58 (Figure 10).

2.6 Card types and addressing

SDSC, SDHC, SDXC

Question: why does a 2 GB card need the block number multiplied by 512, while a 16 GB card does not?
Concept: the first SD cards (*standard capacity*, SDSC, up to 2 GB) took the read and write address in *bytes*. With larger cards, 32-bit byte addresses would run out at 4 GB, so *high capacity* cards (SDHC, up to 32 GB) and *extended capacity* cards (SDXC, up to 2 TB) take a *block number*. The host learns which kind it has from the CCS bit of the OCR register (CMD58).
Example: block 128 is address $128 \cdot 512 = 0x00010000$ on the author's SDSC card, but address `0x00000080` on an SDHC card.

3 The FAT32 file system

3.1 Why a file system?

To the host, the card is only a row of numbered blocks. A *file system* adds the organization that a PC expects: named files of any length, directories, and a record of which blocks are free. FAT (*file allocation table*) was created for floppy disks and grew into FAT12, FAT16 and FAT32, which differ mainly in the width of the table entries (Microsoft Corporation, 2000). Practically every PC, camera and phone reads FAT32, which is why it is the natural choice for a data logger. The definitive description is Microsoft's FAT32 specification (Microsoft Corporation, 2000).

3.2 From the whole card to a single file

Figure 4 shows the levels that the programs walk through, from the first block of the card down to the bytes of a file.

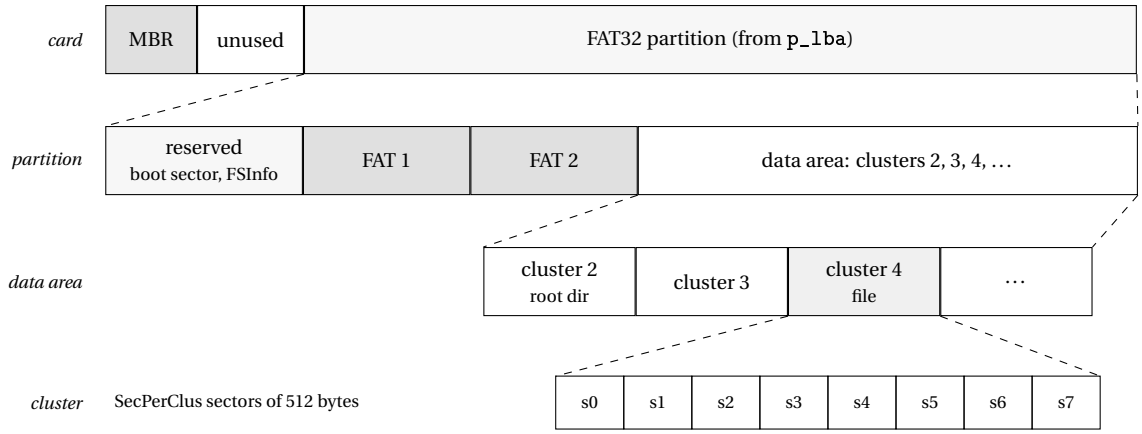


Figure 4: The four levels of a FAT32 card: card, partition, data area, cluster.

Master boot record (MBR)
Block 0 of the card holds a table of up to four partitions; each entry gives the partition type (`0B` or `0C` for FAT32) and its first block (block B4).

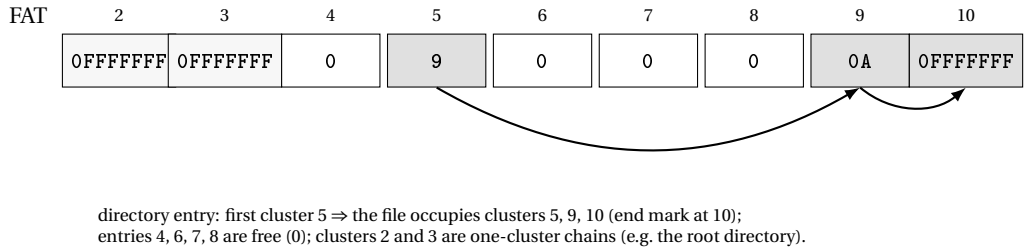
Boot sector and BPB
The first block of the partition contains the *BIOS parameter block*: bytes per sector, sectors per cluster, the number of reserved sectors, the number and size of the FATs, and the first cluster of the root directory (block B4, Table 4). The reserved area also holds the FSInfo sector and a backup copy of the boot sector.

File allocation table (FAT)
One 32-bit entry per cluster; usually two identical copies (block B5).

Data area
Divided into *clusters* of SecPerClus sectors; the first one is cluster number 2. Directories and files are stored in clusters.

3.3 The FAT: a linked list in a table

The FAT is the heart of the system. Its entry number c belongs to cluster c and contains the number of the next cluster of the same file — the clusters of a file form a *linked list* whose links are stored in the table instead of in the clusters themselves (Figure 5). Special values mark free clusters (0) and the end of a chain (`0FFFFFF8` and above); only the lower 28 bits are used (Microsoft Corporation, 2000).



directory entry: first cluster 5 ⇒ the file occupies clusters 5, 9, 10 (end mark at 10);
entries 4, 6, 7, 8 are free (0); clusters 2 and 3 are one-cluster chains (e.g. the root directory).

Figure 5: A file whose clusters are not contiguous: the FAT links 5 → 9 → 10.

3.4 Directories and directory entries

A directory is a file whose content is a list of 32-byte *entries* (Figure 6). The root directory starts at cluster RootClus. Every file has a *short name* of at most eight characters plus a three-character extension (8.3, upper case, padded with spaces, without the dot). Longer names are stored in extra entries with attribute `0F`, placed in front of the short entry; this project reads and writes only the short entries. A first byte `E5` marks a deleted entry, `00` the end of the directory.

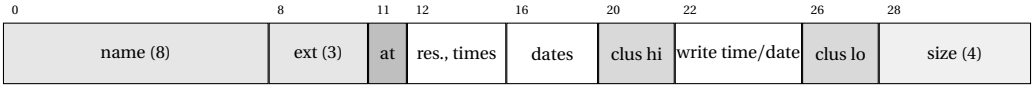


Figure 6: A 32-byte directory entry with byte offsets: name, extension, attribute, time stamps, first cluster (split into a high and a low word) and size in bytes.

3.5 What the programs leave out

FAT32 has more features than a data logger needs. Not implemented here: long file names, subdirectories (other than reading their entries), extending a directory by a new cluster, deleting files, the FSInfo free-cluster hint, and real time stamps. Each of them could be added as a further building block.

4 From byte to file: the layers of the programs

The test programs solve the access in *layers*; each layer uses only the one below it, and each building block adds one layer (Figure 7).

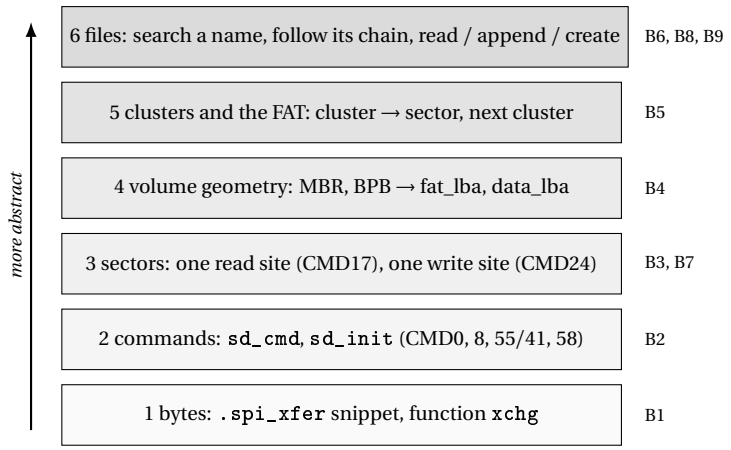


Figure 7: The six layers of the test programs and the blocks that introduce them.

Three ideas run through all programs:

- **Two buffer halves.** A sector is held in `buf_lo` (bytes 0–255) and `buf_hi` (bytes 256–511). An offset o is found in half $\lfloor o/256 \rfloor$ at index $o \bmod 256$.
- **Little-endian numbers.** All multi-byte values on the card are stored least significant byte first. The function `mk32` assembles them into a `raw32`; the subroutine `put_hex32` splits them again.
- **One read site, one write site.** From block B5 on, the main loop is a *state machine*: each pass performs at most one sector access, and the state decides which sector and what to do with it. This keeps the transfer code in one place and makes every step visible in the UART output.

5 Hardware

5.1 The SPI connection

PIC18F452 pin	Function	Direction	SD card (SPI mode)
RC3	SCK	out	CLK
RC5	SDO	out	DI (MOSI, CMD)
RC4	SDI	in	DO (MISO, DAT0)
RC2	CS	out	CS (DAT3), active low

The UART (RC6/RC7, 9600 Bd) carries the debug output; the LEDs sit on RA0 (green) and RA1 (red), as in `18_hello_world`.

5.2 Level shifting

The SD card is a 3.3 V device, the PIC runs at 5 V. Following the author's standing rule for mixed-voltage designs, the lines must be level-shifted: the three PIC outputs SCK, MOSI and CS down to 3.3 V (done on the module, see below), and the card's output MISO must reach a valid high level at the PIC.

Nota Bene 5.1. *RC4/SDI of the PIC18F452 has a Schmitt-trigger (ST) input buffer (Microchip Technology Inc., 2006, Table 9-5, p. 94). For RC3 and RC4 the minimum input high voltage is $V_{IH} = 0.7\,V_{DD}$ (Microchip Technology Inc., 2006, parameter D041, p. 265), i.e.* 3.5 V *at* $V_{DD} = 5.0$ V *and* 3.64 V *at* 5.2 V. *A* 3.3 V *MISO signal therefore lies* below *the guaranteed threshold: it may work on the bench, but outside the specification and with no margin — exactly the situation that made the receiver of the author's water-level telemetry fail at* 5.2 V. *(An ATmega328P at* 5 V *accepts* 3.3 V *as high, which is why such modules are sold as "Arduino-compatible".)*

5.3 The AZ-Delivery micro-SD module

The module used here (AZ-Delivery "SPI reader micro memory card module") has six pins (CS, SCK, MOSI, MISO, VCC, GND) and is specified for a 5 V supply. Its quick-start guide shows a 3.3 V regulator (SOT-223)

and a 14-pin buffer IC, but no schematic. On the author's module the buffer is marked `VHC125`, a 74VHC125 with four buffers. Its 5 V tolerance concerns the supply and the three *inputs* of the card (SCK, MOSI, CS), which the buffer shifts down. The card's *output* MISO, whether it passes the buffer (supplied with 3.3 V) or goes straight to the pin, is expected at about 3.3 V.

The two directions therefore differ:

- **PIC → card** (SCK, MOSI, CS at 5 V): within specification. The buffer on the module is supplied with 3.3 V, and an input protection circuit of the 74VHC125 allows 0 V to 5.5 V at its inputs regardless of the supply voltage (onsemi, 2025, p. 1); shifting these three lines down is exactly the buffer's purpose.
- **Card → PIC** (MISO, at most 3.3 V): below the guaranteed V_{IH} of RC4 (Nota Bene above). Not harmful, but not guaranteed to be read correctly.

Decision. During the bench tests (blocks B1–B7, 2026-10-16) the module was used as it is, and every transfer worked — the actual switching threshold of RC4 is usually well below the guaranteed one. For permanent operation, however, the author decided to add an up-shifter on MISO (Figure 8): a 74HCT125 gate supplied with 5 V (a single-gate 74AHCT1G125 serves as well). A 74HCT input accepts 2.0 V as high, its output swings to 5 V; its output enable $\overline{\text{OE}}$ is tied to CS, so the gate releases RC4 whenever the card is deselected and further SPI devices can share the bus. The software is not affected.

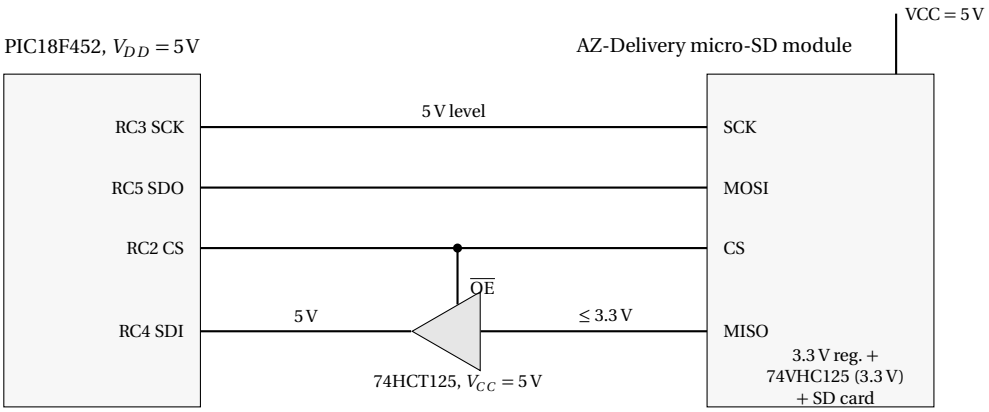


Figure 8: Connection of the SD module for permanent operation: the three PIC outputs go directly to the module (shifted down there), MISO is shifted up by a 74HCT125 gate enabled by CS.

6 Building block B1: the SPI bus

SPI (Serial Peripheral Interface)

Question: how can a microcontroller exchange bytes with a peripheral chip over a few wires, quickly and without a fixed baud rate?

Concept: SPI is a synchronous serial bus. The *master* generates the clock SCK; with every clock pulse one bit leaves the master on MOSI and, at the same time, one bit of the slave arrives on MISO. After eight clocks the master's byte and the slave's byte have been *exchanged* — SPI is always full duplex. A separate chip-select line (CS, active low) chooses the slave. The four *SPI modes* fix the idle level of the clock (CPOL) and the clock edge on which data are sampled (CPHA); SD cards use mode 0 (clock idles low, sample on the rising edge). On a PIC the mode is set with the clock polarity bit CKP and the clock edge bit CKE; which combination a slave needs must be taken from its specification (Layton, 2018, p. 10).

Example: to *read* a byte, the master simply sends the dummy byte `0xFF` and keeps what comes back.

6.1 What this block adds

The PIC18F452 contains the MSSP module (Master Synchronous Serial Port), which implements SPI in hardware; its control registers SSPSTAT and SSPCON1 are described in Section 15 of the data sheet (Microchip Technology Inc., 2006, p. 126). Block B1 adds four snippets to the MyBug PIC18 library:

Snippet	Purpose
`.chip_spi_pins()`	chip part: RC3 = SCK out, RC4 = SDI in, RC5 = SDO out
`.configure_spi(Speed)`	master, mode 0 (CKP = 0, CKE = 1, SMP = 0), clock `SPI_SLOW` (Fosc/64 = 250 kHz), `SPI_MEDIUM` (Fosc/16 = 1 MHz) or `SPI_FAST` (Fosc/4 = 4 MHz)
`.spi_set_speed(Speed)`	changes the clock only (the card initialization must run at ≤400 kHz, the data transfers later at full speed)
`.spi_xfer(Out,In,ref,Local)`	exchanges one byte: writes `Out` to SSPBUF, waits for the buffer-full flag BF, stores SSPBUF in `In`

The chip-select line is deliberately *not* part of the snippets: it is an ordinary output pin that the user program drives with `setbit`/`clearbit`, so that a later program can address several slaves.

6.2 The test: a loopback without a card

Before a card is connected, the SPI master is tested against itself: a jumper connects SDO (RC5) to SDI (RC4). Every byte sent must then come back unchanged. Figure 9 shows the test program `18_sd_1_spi.jan`.

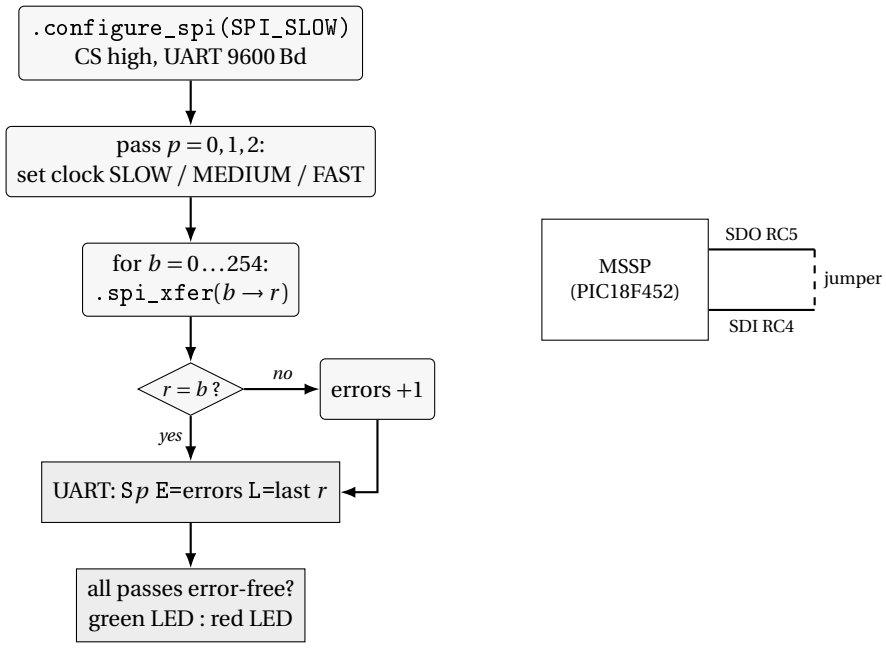


Figure 9: Building block B1: loopback test of the SPI master at three clock rates.

6.3 Program walkthrough

The whole SPI transfer is one snippet of the MyBug library. MyBug snippets are pieces of assembler text with parameters; `$Out`, `$In` and `$ref` are replaced by the variable names and their scope, `$Local` by a unique number for the label.

```
<snippet> .spi_xfer($Out,$In,$ref,$Local)
; full duplex = sends the byte $Out and stores the byte received meanwhile in $In
; ($Out and $In may be the same variable). Blocks for 8 SPI clocks.
   #BANKSEL $Out_$ref
   #MOVF $Out_$ref,W,1
   #MOVWF SSPBUF,0          ;writing SSPBUF starts the 8 clocks
   #spi_busy_$Local:
   #BTFSS SSPSTAT,BF,0
   #GOTO spi_busy_$Local
   #MOVF SSPBUF,W,0         ;received byte, clears BF
   #BANKSEL $In_$ref
   #MOVWF $In_$ref,1
</snippet>
```

- `MOVWF SSPBUF,0` — writing the transmit buffer starts the eight clock pulses; the MSSP shifts the byte out on SDO and, at the same time, the incoming byte in from SDI.
- `BTFSS SSPSTAT,BF,0` / `GOTO` — wait until the buffer-full flag BF says that eight bits have arrived.
- `MOVF SSPBUF,W,0` — reading SSPBUF fetches the received byte and clears BF for the next transfer.

The test program uses the snippet in a double loop: three clock rates, and for each the bytes 0 to 254.

```
    take pass from [0,3) {
       if (pass=0) { .spi_set_speed(SPI_SLOW) }
       altif (pass=1) { .spi_set_speed(SPI_MEDIUM) }
       default { .spi_set_speed(SPI_FAST) }

       errs:=0
       take tx_val from [0,255) {
          .spi_xfer(tx_val,rx_val,context,local_id)
          if (rx_val=tx_val) { }
          default { errs:=errs+1 }
       }
       fails:=fails+errs
```

The empty `if { }` branch with a `default` counts the mismatches; MyBug also has the operator `!=`, which later programs use.
Expected UART output with the jumper in place (one line per clock rate, then a pause of one second):

```
S0 E=00 L=FE
S1 E=00 L=FE
S2 E=00 L=FE
```

Without the jumper, SDI floats or reads a constant level, and errors must appear — a negative test proving that the positive result is real.

6.4 Verification

- The snippet expansions were assembled with MPASM (`/p18F452`) without errors or warnings (2026-10-15); register addresses SSPCON1 = 0xFC6, SSPSTAT = 0xFC7, SSPBUF = 0xFC9 and SSPEN = bit 5 confirmed in the listing.
- Compiled with MyBug and tested on the PIC18F452 board (16 MHz) with the loopback jumper: all three clock rates pass (2026-10-16). The SPI master snippets are thus verified on the hardware and serve as the foundation for block B2.

[figure to be inserted]
Scope trace of SCK and SDO at SPI_SLOW (optional)

7 Building block B2: card initialization

SPI mode of an SD card

Question: an SD card normally talks over its own 4-bit SD bus — how does a small microcontroller with only an SPI port get access to it?

Concept: every SD card also understands a simpler *SPI mode*. After power-up the card is in SD mode; it switches to SPI mode when it receives the reset command CMD0 while its CS line is low. From then on the host sends *command frames* of six bytes, and the card answers each one with a status byte, the *R1 response*. In R1, bit 0 (“in idle state”) is set while the card is still initializing, and bits 1–6 flag errors; bit 7 is always 0, which lets the host recognize the answer among the `0xFF` bytes the idle MISO line delivers.

Example: R1 = `0x01` means “no error, still idle”; R1 = `0x00` means “ready”; R1 = `0x05` means “idle + illegal command”.

7.1 The command frame

A command frame consists of the byte `0x40` + n (start bits `01` plus the 6-bit command index n), a 32-bit argument sent MSB first, and a byte holding a 7-bit CRC followed by the end bit 1. In SPI mode the card checks the CRC only for CMD0 and CMD8; the program nevertheless sends the correct value for every command. All CRC bytes in Table 3 were computed with the CRC-7 polynomial $x^7 + x^3 + 1$ and agree with the values commonly quoted for CMD0 (`0x95`) and CMD8 (`0x87`).

Command	First byte	Argument	CRC byte	Answer	Expected
CMD0 (GO_IDLE_STATE)	64	`00000000`	149 (`0x95`)	R1	`01`
CMD8 (SEND_IF_COND)	72	`000001AA`	135 (`0x87`)	R1 + 4 bytes	`01`, `000001AA`
CMD55 (APP_CMD)	119	`00000000`	101 (`0x65`)	R1	`01`
ACMD41 (SD_SEND_OP_COND)	105	`40000000`	119 (`0x77`)	R1	`01` … `00`
CMD58 (READ_OCR)	122	`00000000`	253 (`0xFD`)	R1 + 4 bytes	`00`, OCR

Table 3: Commands of block B2 (first bytes and CRCs in decimal, as written in the program). For an old V1 card ACMD41 is sent with argument `00000000`, CRC 229.

7.2 The initialization sequence

Figure 10 shows the sequence implemented in `18_sd_2_init.jan`. It runs at `SPI_SLOW` (250 kHz), because the card must be clocked at no more than 400 kHz until it has finished its initialization.

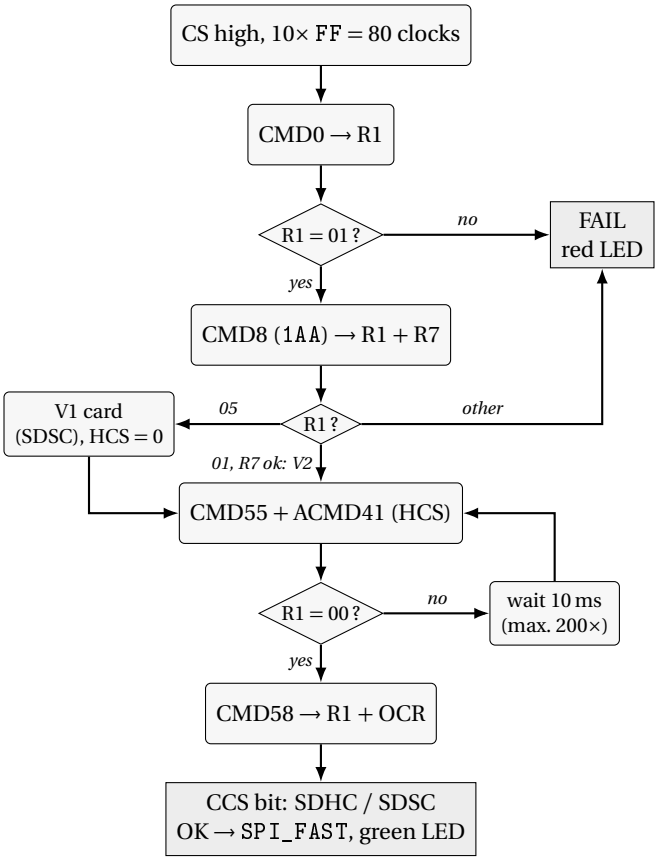


Figure 10: Building block B2: from power-up to an initialized card. Every answer is also printed over the UART; the whole sequence repeats every two seconds.

Three program parts carry the work:

- `xchg(b)` — a MyBug function around `.spi_xfer`: one byte out, one byte back.
- `sd_cmd(cmd,a3,a2,a1,a0,crc)` — sends one idle byte, the six frame bytes, and then reads up to

ten bytes until one has bit 7 cleared (the R1 response arrives after one to eight bytes); it returns R1, or `FF` if the card stays silent.

- `sd_deselect()` — raises CS and sends eight more clocks, after which the card releases MISO.

The chip-select line stays low across CMD55 and ACMD41, which belong together.

OCR and CCS

Question: after initialization, how does the host know whether it must address the card by byte or by block?

Concept: the 32-bit *Operation Conditions Register* (OCR), read with CMD58, holds the supported voltage window and two status bits: bit 31 ("power-up done") and bit 30, *Card Capacity Status* (CCS). CCS = 1 marks an SDHC/SDXC card, whose read and write commands take a *block number* (512-byte units); CCS = 0 marks a standard capacity card (SDSC), addressed in bytes.

Example: OCR = `C0FF8000`: first byte `C0` = bits 31 and 30 set ⇒ initialized SDHC card.

7.3 Program walkthrough

Block B2 introduces the two helpers that every later program contains. The function `xchg` wraps the SPI snippet so that a byte exchange can be written as an expression, `r:=xchg(255)`. The function `sd_cmd` sends one command frame and waits for the R1 answer:

```
61  ; send one SD command frame (6 bytes) and return its R1 answer.
62  ; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
63  ; (CRC only checked by the card for CMD0 and CMD8, sent correctly for all anyway)
64  function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
65     raw8 r,n
66     r:=xchg(255)                    ;one idle byte = 8 clocks, card gets ready
67     r:=xchg(cmd)
68     r:=xchg(a3)
69     r:=xchg(a2)
70     r:=xchg(a1)
71     r:=xchg(a0)
72     r:=xchg(crc)
73     n:=0
74     loop {                          ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
75        r:=xchg(255)
76        if (r<128) { quit }
77        n:=n+1
78        if (n>=10) { quit }          ;no answer -> r stays FF
79     }
80     return(r)
81  }
82
83  ; CS high, then 8 more clocks so the card releases MISO
84  subroutine sd_deselect() {
85     raw8 r
86     setbit(CSPORT,CS)
87     r:=xchg(255)
88  }
```

- The first `xchg(255)` gives the card eight clocks to get ready before the command starts.
- The six frame bytes follow; the return values are thrown away, because the card sends nothing useful while it receives a command.
- The loop then sends `FF` until a byte with bit 7 cleared arrives — that is R1. After ten tries without an answer the function returns `FF`.
- `sd_deselect` raises CS and sends one more byte: only after these eight extra clocks does the card release the MISO line.

The main loop is a direct translation of Figure 10. The ACMD41 step shows how a command that needs repeating is written:

```
168     ; --- 4. ACMD41 = start initialization, repeat until R1 = 00 -------------
169     if (v2=1) { hcs:=64  crc41:=119 }   ;HCS bit 30 = host supports SDHC
170     default { hcs:=0  crc41:=229 }
171     tries:=0
172     loop {
173        clearbit(CSPORT,CS)
```

```
174         r:=sd_cmd(119,0,0,0,0,101)        ;CMD55 = next command is application specific
175         r:=sd_cmd(105,hcs,0,0,0,crc41)    ;ACMD41
176         sd_deselect()
177         if (r=0) { quit }
178         tries:=tries+1
179         if (tries>=200) { quit }          ;200 x 10 ms = 2 s (spec 1 s)
180         delay(10)
181       }
182       putc("A") putc("C") putc("M") putc("D") putc("_fouR") putc("_onE")
183       put_r1(r)
184       putc("Space") putc("N") putc("Equal")
185       put_hex(tries)
186       put_crlf()
```

CMD55 and ACMD41 are sent with CS low in between, because they form one logical command. Up to 200 tries at 10 ms intervals give the card two seconds, twice the time the specification allows for the initialization.

7.4 Expected output

```
CMD0 R1=01
CMD8 R1=01 R7=000001AA
ACMD41 R1=00 N=xx
CMD58 R1=00 OCR=C0FF8000
SDHC OK
```

N, the number of ACMD41 repetitions (in hex, 10 ms apart), depends on the card. Negative tests: without a card every answer reads `FF` and the red LED lights; removing and re-inserting the card during the two-second pause must lead to a fresh, successful initialization.

7.5 Measuring the MISO level

While the program runs, MISO idles high between the answers (the card sends `0xFF`). The voltage at RC4 measured here decides the level-shifting question of Section 5: it must reach $V_{IH} \geq 0.7\,V_{DD}$, i.e. 3.5 V at $V_{DD} = 5.0\,\text{V}$.

7.6 Verification

- Compiled with MyBug and tested on the PIC18F452 board with the AZ-Delivery module (2026-10-16): the card initializes, every answer matches the expectation, and the negative tests (no card, card swapped during the pause) behave as described. This also confirms the first user program with a six-parameter MyBug function (`sd_cmd`).
- The module works without an additional up-shifter on MISO. Measured MISO high level at RC4: *not yet recorded.*

8 Building block B3: reading one sector

Sector (block)

Question: in which units does an SD card store and deliver its data?

Concept: the card's memory is divided into *sectors* (also called *blocks*) of 512 bytes, numbered from 0. The host always reads or writes whole sectors; a single byte cannot be fetched on its own. Sector 0 is special: on a card formatted by a PC it holds the *Master Boot Record* (MBR) with the partition table, and its last two bytes carry the *boot signature* `55 AA`.

Example: byte 510 of the card is byte 510 of sector 0; byte 1000 is byte $1000 - 512 = 488$ of sector 1.

8.1 What this block adds

Block B2 is now packed into one silent function `sd_init()`, which returns 0 (failed), 1 (SDSC) or 2 (SDHC). Each step runs only while the status variable `st` is still 0, so the function has a single `return` at its end. New is the command CMD17 (READ_SINGLE_BLOCK, first byte 81, CRC 85 = `0x55`). Its argument is the sector number for an SDHC card but the *byte address* for an SDSC card; for sector 0 both are 0, so the distinction can wait until block B4.

8.2 The data packet

After R1 = 00 the card needs some time to fetch the sector; meanwhile MISO stays at FF. Then it sends the *data token* FE, the 512 data bytes and a 16-bit CRC (Figure 11). The program polls for the token up to 50 000 times (well above the typical read access time) and ignores the CRC.

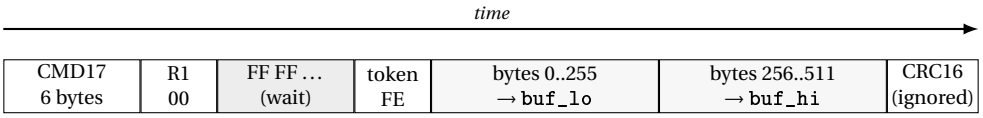


Figure 11: Building block B3: the answer to CMD17 and where the 512 bytes are stored.

8.3 Two halves of 256 bytes

Because PIC18F452 arrays are limited to 256 bytes in MyBug (ISS-004), the sector goes into two arrays, `buf_lo` for bytes 0–255 and `buf_hi` for bytes 256–511. A byte offset o is found in half $\lfloor o/256 \rfloor$ at index $o \bmod 256$; the boot signature at offsets 510 and 511 is therefore `buf_hi[254]` and `buf_hi[255]`.

Nota Bene 8.1. *A* `take k from [0,256)` *loop does not work with a* `raw8` *index: 256 does not fit into eight bits. The program uses the overflow instead:* `k` *starts at 0, is incremented after every byte and wraps from 255 to 0 — the loop ends when* `k = 0` *again, after exactly 256 bytes. The same* `k` *(0 again) then serves the second half.*

8.4 Program walkthrough

From this block on, the initialization of B2 is packed into one function, `sd_init()`, which prints nothing and returns the card type (0 = failed, 1 = SDSC, 2 = SDHC). The new part is the reading of one block:

```
198     ; --- CMD17 = READ_SINGLE_BLOCK, sector 0 --------------------------------
199     mrk:=255
200     if (fail=0) {
201        clearbit(CSPORT,CS)
202        r:=sd_cmd(81,0,0,0,0,85)              ;CMD17, argument 0, CRC 0x55
203        if (r=0) {
204           n:=0                               ;wait for the data token FE
205           loop {                             ;(card may need up to ~100 ms)
206              mrk:=xchg(255)
207              if (mrk!=255) { quit }
208              n:=n+1
209              if (n>=50000) { quit }
210           }
211        }
212        if (mrk=254) {
213           k:=0                               ;bytes 0..255
214           loop {
215              t:=xchg(255)
216              buf_lo[k]:=t
217              k:=k+1
218              if (k=0) { quit }               ;raw8 wrapped after 256 bytes
219           }
220           loop {                             ;bytes 256..511 (k is 0 again)
221              t:=xchg(255)
222              buf_hi[k]:=t
223              k:=k+1
224              if (k=0) { quit }
225           }
226           t:=xchg(255)                       ;CRC16 of the data block, ignored
227           t:=xchg(255)
228        }
229        sd_deselect()
```

- `sd_cmd(81,0,0,0,0,85)` is CMD17 (64 + 17 = 81) with address 0.
- The first loop waits for the start token: as long as the card answers FF it is still fetching the block.
- The two following loops store the 512 bytes. A `raw8` index counts from 0 to 255 and then wraps back to 0, which ends each loop after exactly 256 bytes (Nota Bene above).
- The two bytes after the data are the block's CRC, which SPI mode does not require the host to check.

8.5 Expected output

```
INIT OK SDHC
CMD17 R1=00 TOK=FE
000 FA 33 C0 8E D0 ...
...
1F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA
SIG 55AA OK
```

The exact bytes depend on the tool that formatted the card. The first 446 bytes may contain boot code or zeros; the partition table (offsets 446–509, lines `1B0` to `1F0`) is the subject of block B4. A card formatted without a partition table ("superfloppy") has the FAT32 boot sector in sector 0 instead; it also ends with `55 AA`.

Example 8.1. *The last lines of the dump of the author's card (2026-10-16):*

```
1C0 03 00 0C FE 3F EE 80 00 00 00 80 CF 3A 00 00 00
1D0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1E0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
1F0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 55 AA
```

The first partition entry starts at offset 446 (`1BE`*); its first two bytes (status, start head) are still on line* `1B0`*. On line* `1C0` *follow the start sector/cylinder* `03 00`*, the partition type* `0C` *(FAT32 with LBA addressing), the end CHS address* `FE 3F EE`*, and two 32-bit numbers stored* little-endian *(least significant byte first):*

- *start sector* `80 00 00 00` ⇒ `0x00000080` *= 128,*
- *number of sectors* `80 CF 3A 00` ⇒ `0x003ACF80` *= 3 854 208 sectors =* 1 973 354 496 B ≈ 1.97 GB.

Entries 2 to 4 (lines `1CE`–`1FD`*) are empty. The FAT32 boot sector of this card is therefore sector 128 — the starting point of block B4.*

8.6 Verification

- Compiled with MyBug and tested on the hardware (2026-10-16): sector 0 is read, dumped, and the program reports `SIG 55AA OK`. This also confirms the first MyBug program with two 256-byte arrays on the PIC18F452, the raw8-wrap loops and a constant index into an array outside bank 0.
- Cross-check: the dump of sector 0 should be compared with the same sector read on a PC (e.g. with a hex editor on the raw device).

9 Building block B4: partition table and boot sector

Partition and file system

Question: sector 0 holds a partition table — where do the files begin?
Concept: a card can be split into up to four *partitions*, each a contiguous range of sectors described by a 16-byte entry in the MBR (offsets 446, 462, 478, 494). An entry gives the partition *type* (`0B` or `0C` for FAT32) and its *start sector*, the *logical block address* (LBA). The first sector of a FAT32 partition is its *boot sector*; it carries the *BIOS parameter block* (BPB), a table of numbers that describe the layout of the file system.
Example: on the author's card, partition 1 has type `0C` and starts at LBA 128 (Example in Section 8).

9.1 The layout of a FAT32 partition

A FAT32 partition consists of three areas (Figure 12): the *reserved sectors* (beginning with the boot sector), one or more copies of the *file allocation table* (FAT), and the *data area*, which is divided into *clusters* of *SecPerClus* sectors. The first cluster of the data area is cluster number 2; the root directory is a cluster chain like any file and starts at cluster *RootClus* (usually 2).

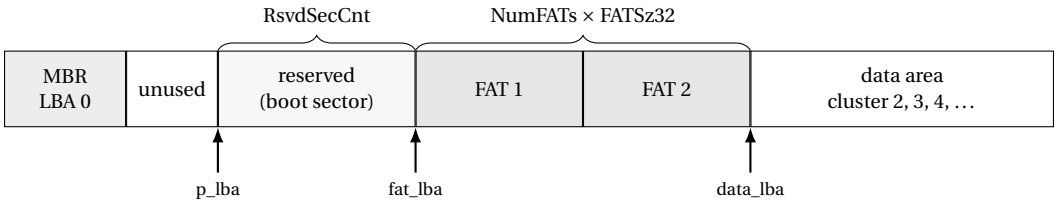


Figure 12: Building block B4: the areas of the card and the three sector numbers computed from the MBR and the BPB.

The two numbers that every later block needs follow directly:

$$fat_lba = p_lba + RsvdSecCnt,$$
$$data_lba = fat_lba + NumFATs \cdot FATSz32.$$

Offset	Field	Size	Meaning
11	BytsPerSec	2	bytes per sector, must be 512 (`0200`)
13	SecPerClus	1	sectors per cluster (1, 2, 4, …, 128)
14	RsvdSecCnt	2	reserved sectors before FAT 1
16	NumFATs	1	number of FAT copies (usually 2)
36	FATSz32	4	sectors per FAT
44	RootClus	4	first cluster of the root directory
82	BS_FilSysType	8	text `"FAT32   "` (informative only)
510	Signature	2	`55 AA`

Table 4: The BPB fields read by block B4 (all multi-byte values little-endian).

9.2 What this block adds

- **One read routine for two sectors.** The program reads sector 0 and then the boot sector with the same code, inside `take step from [0,2)`; the sector number `lba` is set to 0 before the loop and to `p_lba` after the MBR has been evaluated.
- **Card-type addressing.** The sector number becomes the CMD17 argument: for an SDHC card the LBA itself, for an SDSC card the byte address $LBA \cdot 512$. The author's card is an SDSC card: the boot sector at LBA 128 is read from byte address $128 \cdot 512 = 65\,536 = \texttt{0x00010000}$.
- **32-bit values.** The new function `mk32(b3,b2,b1,b0)` assembles a `raw32` from four bytes (each little-endian field is passed most significant byte first). The new subroutine `put_hex32` splits a `raw32` again: `raw8(w)` keeps the low byte, `w/256` drops it. The same split turns the address into the four argument bytes of CMD17.
- **Multiplication by addition.** *NumFATs · FATSz32* is computed by adding `FATSz32` `NumFATs` times (1 or 2 additions) — transparent, and no 32-bit multiplication routine is needed.

Nota Bene 9.1. *Array elements are copied into plain* `raw8` *variables before they are passed to* `mk32`. *Passing* `buf_hi[201]` *directly as a function argument has not been tested in MyBug yet; the detour keeps this block to features already verified.*

9.3 Program walkthrough

Block B4 is the first to compute with 32-bit numbers. Two small routines carry the conversion between the card's little-endian bytes and MyBug's `raw32` type:

```
113 ; (new) four bytes (most significant first) -> one raw32 value
114 function raw32 mk32(raw8 b3,raw8 b2,raw8 b1,raw8 b0) {
115     raw32 v
116     v:=raw32(b3)
117     v:=v*256
118     v:=v+raw32(b2)
119     v:=v*256
```

```
    v:=v+raw32(b1)
    v:=v*256
    v:=v+raw32(b0)
    return(v)
}

; (new) raw32 -> 8 hex digits; raw8(w) keeps the low byte (narrowing), w/256 drops it
subroutine put_hex32(raw32 v) {
    raw32 w
    raw8 b3,b2,b1,b0
    w:=v
    b0:=raw8(w)
    w:=w/256
    b1:=raw8(w)
    w:=w/256
    b2:=raw8(w)
    w:=w/256
    b3:=raw8(w)
    put_hex(b3)
    put_hex(b2)
    put_hex(b1)
    put_hex(b0)
}
```

`mk32` builds the value like a decimal number is read, digit by digit, but in base 256: multiply by 256, add the next byte. `put_hex32` goes the other way: `raw8(w)` keeps only the lowest byte, `w/256` removes it. The same splitting turns a sector number into the four argument bytes of CMD17:

```
            ; --- sector number -> CMD17 argument (4 bytes, MSB first) --------
            adr:=lba
            if (kind=1) { adr:=lba*512 }       ;SDSC = byte address
            a0:=raw8(adr)
            adr:=adr/256
            a1:=raw8(adr)
            adr:=adr/256
            a2:=raw8(adr)
            adr:=adr/256
            a3:=raw8(adr)
```

For an SDSC card the sector number is first multiplied by 512, because these cards expect a byte address (Section 2.6).

9.4 Expected output

```
INIT OK SDSC
MBR SIG OK TYPE=0C LBA=00000080
BPB SIG OK FAT32
BPS=0200 SPC=xx RSV=xxxx NF=02
FSZ=xxxxxxxx ROOT=00000002
FAT=xxxxxxxx DATA=xxxxxxxx
```

The values marked `xx` depend on the formatting tool. They can be checked by hand with the two formulas above, and against a PC: a hex view of the boot sector, or the cluster size and FAT size reported by the formatting tool.

Example 9.1. *Output for the author's card (2026-10-16):*

```
INIT OK SDSC
MBR SIG OK TYPE=0C LBA=00000080
BPB SIG OK FAT32
BPS=0200 SPC=08 RSV=02A8 NF=02
FSZ=00000EAC ROOT=00000002
FAT=00000328 DATA=00002080
```

Checked by hand: fat_lba $= 128 + 680 = 808 =$ `0x328` *and data_lba* $= 808 + 2 \cdot 3756 = 8320 =$ `0x2080` *— both agree with the PIC. Clusters have* $8 \cdot 512 = 4096$ B. *The FAT size is plausible, too: the partition has 3 854 208 sectors, of which* $3\,854\,208 - 680 - 2 \cdot 3756 = 3\,846\,016$ *form the data area, i.e. 480 752 clusters. With 4 bytes per FAT entry (plus the two reserved entries 0 and 1) one FAT needs* $(480\,752 + 2) \cdot 4/512 \approx 3755.9$ *sectors, rounded up to the 3756 (*`0xEAC`*) stored in the BPB. The root directory starts at cluster 2, i.e. at the first sector of the data area, LBA 8320 — the starting point of block B5.*

9.5 Verification

- Compiled with MyBug and tested on the hardware (2026-10-16): all fields are read, and the computed sector numbers agree with the hand calculation above. This confirms `raw32` arithmetic on the PIC18F452 (assembling from bytes, multiplication by 512, addition, division by 256, narrowing with `raw8()`), and SDSC byte addressing beyond sector 0.

10 Building block B5: the root directory

Cluster chain and FAT

Question: a directory or a file may be larger than one cluster — how does the card know which cluster comes next?

Concept: the *file allocation table* (FAT) has one 32-bit entry per cluster. The entry of cluster c holds the number of the *next* cluster of the same file, or an end mark ($\geq$ `0FFFFFF8`). Following these numbers from the first cluster gives the *cluster chain*. Only the lower 28 bits are used; the upper four are reserved and must be masked.

Example: FAT[2] = `0FFFFFFF` means that the root directory (cluster 2) fits into one cluster; FAT[5] = 6, FAT[6] = `0FFFFFFF` is a file occupying clusters 5 and 6.

Directory entry

Question: where are a file's name, size and first cluster stored?

Concept: a directory is a sequence of 32-byte *entries*: the 8.3 name (bytes 0–10, padded with spaces, without the dot), the *attribute* byte (11), the first cluster (high word at 20/21, low word at 26/27) and the size in bytes (28–31). A first byte `00` ends the directory, `E5` marks a deleted entry, and attribute `0F` marks a piece of a *long file name*, which this project skips: every file also has a short 8.3 name (e.g. `LONGFI~1.TXT`).

Example: attribute `20` = archive (a normal file), `10` = subdirectory, `08` = volume label.

10.1 From cluster number to sector number

Cluster c occupies *SecPerClus* sectors starting at

$$LBA(c) = data_lba + (c-2) \cdot SecPerClus.$$

The FAT entry of cluster c lies in sector $fat_lba + \lfloor c/128 \rfloor$ (128 entries of 4 bytes per sector) at byte offset $4 \cdot (c \bmod 128)$. With two 256-byte halves, entries 0–63 of a FAT sector are in `buf_lo`, entries 64–127 in `buf_hi`. For the author's card, the root directory (cluster 2) starts at LBA 8320, i.e. at byte address $8320 \cdot 512 =$ `0x00410000`.

10.2 A state machine with one read

Four kinds of sectors are read now: the MBR, the boot sector, directory sectors and FAT sectors. Instead of four copies of the read code, the main loop reads exactly *one* sector per pass and then evaluates it according to a *state* (Figure 13). Each state computes the sector number for the next pass and chooses the next state.

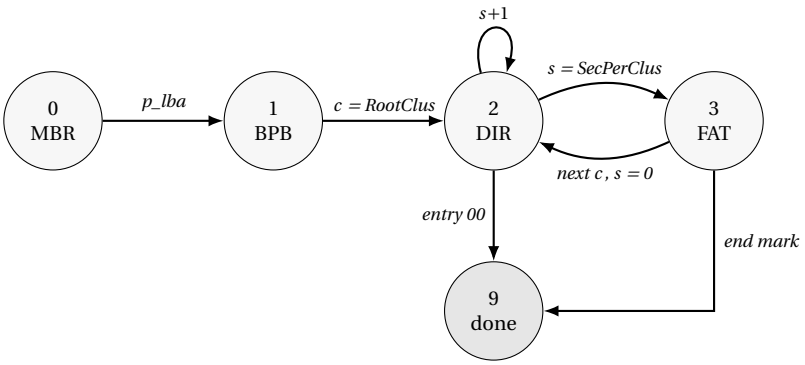


Figure 13: Building block B5: the states of the main loop; every arrow is one sector read.

Nota Bene 10.1. *The state machine also sidesteps an open MyBug question: the two buffer arrays live in main, and whether a subroutine can reach main's* arrays *with the @ marker (proven for scalar variables) has not been tested yet. With a single read site in main, no read subroutine is needed at all.*

10.3 Processing a directory sector

A sector holds 16 entries. Entry e lies in half $\lfloor e/8 \rfloor$ at byte offset $32 \cdot (e \bmod 8)$; the program copies its 32 bytes into the small array `ent[32]` and then reads all fields with constant indices. The safety limit of 200 clusters stops a corrupted, looping chain.

10.4 Program walkthrough

The main loop of B5 has the shape that all later programs share:

1. compute the sector number (for state 2 from cluster and sector),
2. read the sector — the only CMD17 in the program,
3. evaluate the sector according to `state` and choose the next state.

The sector number of a directory sector follows directly from $LBA(c) = data_lba + (c-2) \cdot SecPerClus + s$; the multiplication is done by repeated addition:

```
236         if (state=2) {
237            pm:=c-2
238            lba:=0
239            i:=0
240            loop {                              ;lba = (c-2)*spc by repeated addition
241               if (i>=spc) { quit }
242               lba:=lba+pm
243               i:=i+1
244            }
245            lba:=lba+data_lba
246            lba:=lba+raw32(s)
247         }
```

State 3 finds the FAT entry of cluster c in the sector just read and assembles the next cluster number:

```
406         default {                              ;state 3 = FAT sector, find the next cluster
407            q:=raw8(c)
408            q:=q&127                            ;c mod 128 = entry number in this sector
409            half:=q/64                          ;entries 0..63 in buf_lo, 64..127 in buf_hi
410            idx:=q&63
411            idx:=idx*4                          ;byte offset of the 4-byte entry
412            if (half=0) {
413               c0:=buf_lo[idx]
414               idx:=idx+1
415               c1:=buf_lo[idx]
416               idx:=idx+1
417               c2:=buf_lo[idx]
418               idx:=idx+1
419               c3:=buf_lo[idx]
420            }
421            default {
422               c0:=buf_hi[idx]
423               idx:=idx+1
```

```
            c1:=buf_hi[idx]
            idx:=idx+1
            c2:=buf_hi[idx]
            idx:=idx+1
            c3:=buf_hi[idx]
         }
         c3:=c3&15                          ;FAT32 entries use only 28 bits
         c:=mk32(c3,c2,c1,c0)
         ncl:=ncl+1
         if (c>=268435448) { state:=9 }     ;0FFFFFF8.. = end of the cluster chain
         altif (c<2) { fail:=1 }            ;free or reserved = broken chain
         altif (ncl>=200) { fail:=1 }       ;safety limit against a looping chain
         default {
            s:=0
            state:=2
         }
      }
```

- `q:=raw8(c)` and `q&127` give c mod 128, the entry number inside the FAT sector (the low seven bits of c).
- Entries 0–63 are in `buf_lo`, entries 64–127 in `buf_hi`; the byte offset is four times the entry number within the half.
- `c3&15` removes the four reserved upper bits.
- An end mark finishes the listing; a value below 2 would be a broken chain.

10.5 Expected output

```
ROOT C=00000002
SDCARD     .    A=08 C=00000000 S=00000000 <VOL>
TEST    .TXT A=20 C=00000003 S=0000001A
LOGS    .    A=10 C=00000004 S=00000000 <DIR>
N=03 OK
```

(The names are examples; the list shows what is on the card. Hidden system entries such as `SYSTEM~1` written by Windows appear as well.) The listing is checked against the directory shown by a PC.

Example 10.1. *The root directory of the author's card (2026-10-16):*

```
ROOT C=00000002
XX      .    A=08 C=00000000 S=00000000 <VOL>
SYSTEM~1.    A=16 C=00000003 S=00000000 <DIR>
TEST02  .TXT A=20 C=00000005 S=00000047
DATA2   .TXT A=20 C=00000006 S=000001CA
DATA    .TXT A=20 C=00000007 S=00000C5E
DATA_V~3.TXT A=20 C=0000002B S=00000201
DATA_V~1.TXT A=20 C=00000008 S=00000403
DATA_V~2.TXT A=20 C=0000000B S=0001F09E
DATA_A~1.TXT A=20 C=0000002C S=000001CA
DATA_A~3.TXT A=20 C=0000002D S=00000C18
DATA_A~4.TXT A=20 C=0000002E S=00006616
DATA_A~2.TXT A=20 C=00000035 S=00001CA7
DA29C5~1.TXT A=20 C=00000037 S=00019D10
DATA_A~5.TXT A=20 C=00000051 S=00000558
N=0E OK
```

The volume label is `XX`; `SYSTEM~1` *with attribute* $16 = \text{directory} + \text{system} + \text{hidden}$ *is Windows' "System Volume Information" folder; the names with* `~` *are the short names of files with long names, whose long-name entries were skipped as intended.*

The listing can be checked against itself. Files written one after the other on an empty card occupy consecutive clusters, so a file of S bytes that starts at cluster c should be followed by the next file at $c + \lceil S/4096 \rceil$. For instance, `DATA_V~2.TXT` *has* $\texttt{0x1F09E} = 127\,134$ *bytes, i.e. 32 clusters:* $\texttt{0x0B} + 32 = \texttt{0x2B}$*, exactly where* `DATA_V~3.TXT` *begins. The same holds for the chain* $\texttt{0x2B} \to \texttt{0x2C} \to \texttt{0x2D} \to \texttt{0x2E} \to \texttt{0x35} \to \texttt{0x37} \to \texttt{0x51}$ *(sizes of 1, 1, 1, 7, 2 and 26 clusters).*

10.6 Verification

- Compiled with MyBug and tested on the hardware (2026-10-16): the root directory is listed completely (14 entries, the volume label and the hidden system folder included), and the sizes and first clusters are consistent with each other.
- State 3 was not exercised here: the root directory of this card ends inside its first cluster, so the FAT lookup never ran. It was tested in block B6 by reading `DATA_V~2.TXT`, which spans 32 clusters.

11 Building block B6: reading a file

File = directory entry + cluster chain

Question: block B5 found the names — what does it take to get at a file's content?

Concept: the directory entry supplies the first cluster and the exact size in bytes; the FAT supplies the order of all further clusters. Reading a file therefore means reading the clusters of its chain one after the other and stopping after *size* bytes, because the last cluster is usually only partly used.

Example: `DATA_V~2.TXT` (127 134 bytes) starts at cluster `0x0B` and occupies 32 clusters of 4096 bytes; of the last cluster only $127\,134 - 31 \cdot 4096 = 158$ bytes belong to the file.

11.1 What this block adds

- **Search by name.** The 8.3 name is stored as 11 bytes without the dot, exactly as in the directory entry: `target = "DATA_V~2TXT"`. State 2 compares each entry byte by byte; long-name pieces and directories never match.
- **A new state 4 (file sector).** It sends min(512, *rem*) bytes of the sector and decreases the remaining byte count `rem`; at $rem = 0$ the file is complete.
- **State 3 serves both.** The FAT lookup is the same for the directory and for the file; the flag `fileon` decides whether it continues with state 2 or state 4. This block is the first to exercise state 3 on the hardware.
- **Faster UART.** At 9600 Bd the 127 134 bytes would take over two minutes; the program uses 38 400 Bd (SPBRG = 25 at 16 MHz, error 0.16 %), about 33 s.

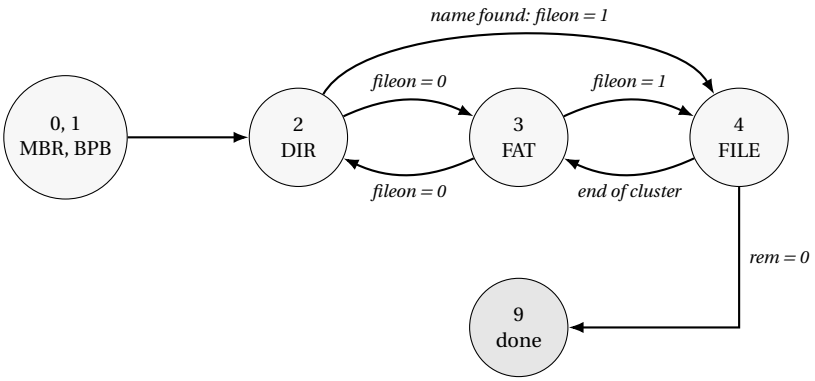


Figure 14: Building block B6: the state machine of B5 with the new file state.

11.2 Checking the transfer

Watching 127 134 characters scroll by proves little. At the end the program therefore reports the number of bytes sent, a 16-bit sum of all bytes and the number of clusters read. The same two numbers are computed on the PC from the original file, for example with Python:

```
d = open("DATA_V~2.TXT", "rb").read()
print(hex(len(d)), hex(sum(d) & 0xFFFF))
```

Equal sums do not prove every byte, but a lost, doubled or swapped sector would change them almost certainly; and `N` must equal the size from the directory entry.

11.3 Program walkthrough

Two pieces are new. The name comparison in state 2 runs over the 11 bytes of the entry and of the array `target`:

```
370           take j from [0,11) {              ;compare the 11 name bytes
371              t:=ent[j]
372              u:=target[j]
373              if (t!=u) { match:=0 }
374           }
375           attr:=ent[11]
376           if (attr=15) { match:=0 }         ;never a long-name piece
377           t:=attr&16
378           if (t!=0) { match:=0 }            ;never a directory
```

State 4 sends the content of a file sector, but never more than the bytes still missing (`rem`):

```
418     altif (state=4) {                       ;file sector, send min(512, rem) bytes
419        k:=0
420        loop {                               ;bytes 0..255
421           if (rem=0) { quit }
422           t:=buf_lo[k]
423           putc(t)
424           sum:=sum+raw16(t)
425           rem:=rem-1
426           nsent:=nsent+1
427           k:=k+1
428           if (k=0) { quit }
429        }
430        k:=0
431        loop {                               ;bytes 256..511
432           if (rem=0) { quit }
433           t:=buf_hi[k]
434           putc(t)
435           sum:=sum+raw16(t)
436           rem:=rem-1
437           nsent:=nsent+1
438           k:=k+1
439           if (k=0) { quit }
440        }
441        if (rem=0) { state:=9 }
442        default {
443           s:=s+1
444           if (s>=spc) {
445              pm:=c/128
446              lba:=fat_lba+pm
447              state:=3
448           }
449        }
450     }
```

After the last sector of a cluster the program switches to state 3, which now continues with state 4 instead of state 2 because the flag `fileon` is set.

11.4 Expected output

```
FOUND C=0000000B S=0001F09E
----
<content of DATA_V~2.TXT>
----
END N=0001F09E SUM=xxxx CL=20
```

`CL=20` (hex) = 32 clusters. The program runs once; reset to repeat.

Nota Bene 11.1. *The mode flag was first called* `infile`. *MyBug rejected the declaration line with "variable or subroutine does not exist". The cause, confirmed by the author of MyBug: the tokenizer recognizes* Inf *and* NaN *as number constants — prepared for a future floating-point type — and therefore reads an identifier beginning with* `inf` *(here in lower case) as the number infinity, leaving the rest* `ile` *as an unknown name. It is a relative of the known rule that no keyword may appear inside an identifier. Renamed to* `fileon`*; the rule is recorded in MyBug's HOW-TO list (HOWTO-021).*

Example 11.1. *The end of the transfer on the author's card (2026-10-16); the file is a log of voltage readings:*

```
Voltage: 0.2989563V
Voltage: 0.299762V
Voltage: 0.2989563V

----
END N=0001F09E SUM=510E CL=20
```

`N = 0x1F09E = 127 134` *bytes, exactly the size from the directory entry, and* `CL = 0x20 = 32` *clusters, exactly* [127 134/4096].

11.5 Verification

- Compiled with MyBug (after renaming `infile`, see the Nota Bene above) and tested on the hardware (2026-10-16): the file is found and transferred completely; byte count and cluster count agree with the directory entry. This is the first hardware run of the FAT lookup (state 3): 31 lookups, and a single wrong link would have broken the chain or changed the count.
- 16-bit sum `510E`: comparison with the sum computed on the PC *pending*.

12 Building block B7: writing one sector

Write cycle of a flash card

Question: reading is answered at once — why does writing need more steps?

Concept: an SD card stores its data in flash memory, which must be erased and programmed internally. After receiving a sector the card therefore (1) confirms the reception with a *data response* byte and then (2) stays *busy* for a while, signalling this by holding MISO low (the host reads `00`). Only when MISO is high again may the next command follow.

Example: data response `xxx00101`, i.e. low five bits `05` = "data accepted"; `0B` = CRC error, `0D` = write error.

12.1 A safe test sector

This block is the first one that changes the card. To leave the file system and all files untouched, the test uses a sector that belongs to no partition: the last sector of the gap between the MBR and the partition start, $tlba = p_lba - 1$ (LBA 127 on the author's card, byte address $127 \cdot 512 =$ `0xFE00`). Two rules make the test harmless:

- it writes only if the test sector is completely zero (otherwise it stops with `NOT EMPTY` before the first write), and
- it writes zeros back at the end and checks them.

12.2 The CMD24 sequence

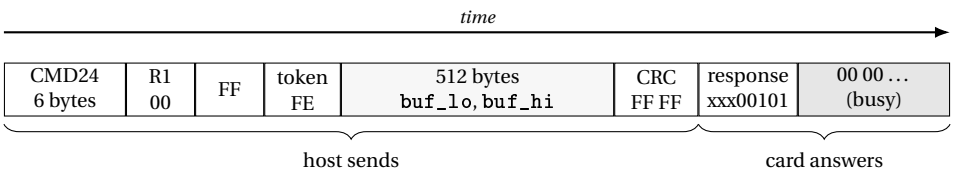


Figure 15: Building block B7: one CMD24 write (R1 is the card's answer to the command).

12.3 Six operations, one read and one write site

As in block B5, the program avoids copies of the transfer code. It runs six operations in `take op from [0,6)`; operations 2 and 4 go to the write site, all others to the read site:

op	access	sector	purpose
0	read	0	MBR → `p_lba`, *tlba* = *p_lba* − 1
1	read	tlba	must be all zero, else stop
2	write	tlba	pattern: `buf_lo[k]` = *k*, `buf_hi[k]` = 255 − *k*
3	read	tlba	compare with the pattern → `ERR`
4	write	tlba	zeros (restore)
5	read	tlba	must be all zero again

The two halves get different patterns, so swapped halves would be detected; before operation 3 the buffer is cleared, so a failed read cannot pass for a successful one. A verify error does not stop the program: the zeros are still written back.

12.4 Program walkthrough

The write site mirrors the read site of block B3:

```
279         if (wr=1) {
280            ; --- the ONE write site = CMD24 ----------------------------------------
281            wres:=255
282            busy:=0
283            clearbit(CSPORT,CS)
284            r:=sd_cmd(88,a3,a2,a1,a0,255)       ;CMD24 = WRITE_BLOCK
285            if (r=0) {
286               t:=xchg(255)                     ;one byte gap before the token
287               t:=xchg(254)                     ;start token FE
288               k:=0
289               loop {
290                  u:=buf_lo[k]
291                  t:=xchg(u)
292                  k:=k+1
293                  if (k=0) { quit }
294               }
295               loop {
296                  u:=buf_hi[k]
297                  t:=xchg(u)
298                  k:=k+1
299                  if (k=0) { quit }
300               }
301               t:=xchg(255)                     ;CRC16, not checked by the card
302               t:=xchg(255)
303               wres:=xchg(255)                  ;data response xxx0sss1
304               wres:=wres&31                    ;05 = data accepted
305               loop {                           ;busy, the card holds MISO low
306                  t:=xchg(255)
307                  if (t!=0) { quit }
308                  busy:=busy+1
309                  if (busy>=60000) { quit }
310               }
311            }
312            sd_deselect()
313            putc("W") putc("R") putc("I") putc("T") putc("E") putc("Space")
314            putc("R") putc("Equal")
315            put_hex(wres)
316            putc("Space") putc("B") putc("U") putc("S") putc("Y") putc("Equal")
317            t:=high(busy)
318            put_hex(t)
319            t:=low(busy)
320            put_hex(t)
321            put_crlf()
322            if (wres!=5) { fail:=1 }
323            if (busy>=60000) { fail:=1 }
324         }
```

- `sd_cmd(88,...)` is CMD24 (64 + 24 = 88).
- One `FF` byte separates the command answer from the start token `FE`.
- The 512 data bytes are sent with `xchg(u)`; what comes back is ignored.
- `wres&31` keeps the five low bits of the data response; `05` means “accepted”.
- The busy loop counts the polls until MISO is no longer `00`.

12.5 Expected output

```
INIT OK SDSC
TEST LBA=0000007F
ZERO OK
WRITE R=05 BUSY=xxxx
VERIFY ERR=0000
WRITE R=05 BUSY=xxxx
RESTORE OK
```

`BUSY` counts the polls until the card released MISO; it shows how long the card needs for one sector.

Example 12.1. *Output on the author's card (2026-10-16):*

```
INIT OK SDSC
TEST LBA=0000007F
ZERO OK
WRITE R=05 BUSY=1249
VERIFY ERR=0000
WRITE R=05 BUSY=0236
RESTORE OK
```

Both writes were accepted (`05`*), all 512 bytes came back unchanged, and the sector is empty again. Remarkably, the first write kept the card busy for* $\texttt{0x1249} = 4681$ *polls, the second (same sector) only for* $\texttt{0x236} = 566$ *— about eight times less. The duration of one poll was not measured, so the program gives only the ratio; a plausible reason is internal work of the card's controller (erasing or reassigning a flash block) on the first write. Block B8 will show whether this is typical.*

12.6 Verification

- Compiled with MyBug and tested on the hardware (2026-10-16): CMD24 works, the data response is `05`, the pattern is read back without error and the test sector is restored to zeros. The card can now be written.

13 Building block B8: appending to a file

Read-modify-write

Question: the card can only write whole sectors — how are 29 bytes added behind the 71 bytes of a file?
Concept: the sector that holds the end of the file is *read* into the buffer, the new bytes are placed behind the old ones (*modify*), and the whole sector is *written* back. The same pattern changes a single FAT entry (4 bytes) or a single directory entry (32 bytes) inside a sector. Only a sector that starts empty needs no read: it is filled with zeros instead.
Example: `TEST02.TXT` (71 bytes, cluster 5) ends in sector $8320 + (5-2) \cdot 8 = 8344$; bytes 71–99 of that sector receive the new line.

13.1 What has to change on the card

Appending touches up to three places, and their order matters:

1. **the data** — the new bytes in the last sector(s) of the file;
2. **the FAT** — only if the last cluster is full: a free cluster (entry value 0) is marked as end of chain, and the entry of the old last cluster is set to the new cluster number; both FAT copies are written;
3. **the directory entry** — the new size (and, for a file that was empty, its first cluster).

The directory entry is written last. If the power fails before, the file keeps its old size and the new bytes are simply invisible; at worst a cluster is marked as used without belonging to a file (a "lost cluster", which `chkdsk` reclaims) — but the existing content is never damaged.

Nota Bene 13.1. *Two things a PC would also update are deliberately left alone: the free-cluster count in the FSInfo sector and the modification time of the entry (the PIC has no clock yet). The FSInfo values (free-cluster count and next free cluster) are only* hints*; an implementation must not rely on them, and the value* `FFFFFFFF` *means "unknown" (Microsoft Corporation, 2000). On the author's card,* `chkdsk` *reported no errors after the PIC had appended to and created files.*

13.2 The state machine

Block B8 extends the state machine of B5/B6 by a write site (as in B7) and by states that do not access the card at all. Each pass of the main loop has three parts: (A) *prepare* — the state chooses the sector number and the access (`io` = none, read or write); (B) the single read or write site; (C) *evaluate* — the state works on the buffer and chooses the next state.

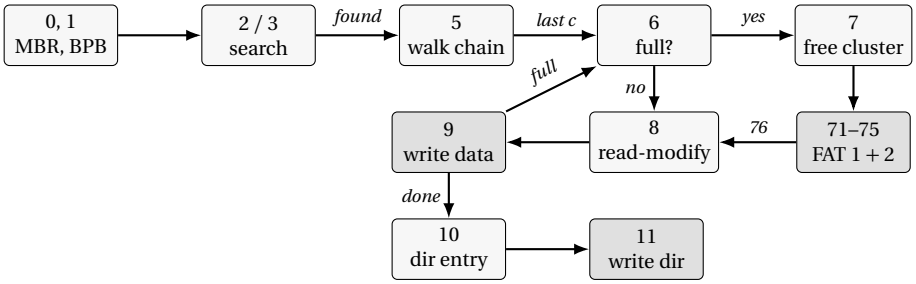


Figure 16: Building block B8: the main states (dark = write access). From state 9 the machine also returns to state 8 while bytes remain in the same cluster.

13.3 Where the end of the file is

Walking the chain (state 5) counts the clusters *nclus* of the file and stops at the last one. The bytes already used in it are

$$coff = fsize - (nclus-1)\cdot cbytes, \qquad 1 \le coff \le cbytes = SecPerClus \cdot 512,$$

and $coff = cbytes$ means "full". In state 8 the sector inside the cluster is $\lfloor coff/512 \rfloor$ and the byte inside the sector $bo = coff \bmod 512$; the half of the buffer is the high byte of `bo` and the index its low byte.

13.4 Allocating a cluster

State 7 reads the FAT sector by sector (from the beginning) and takes the first entry whose 28 bits are all zero, skipping the reserved clusters 0 and 1 and any number beyond the last cluster of the data area, $maxc = (TotSec32 - (data_lba - p_lba))/SecPerClus + 1$. It writes the end mark `0FFFFFFF` into that entry, keeping the four reserved upper bits, and stores the sector into FAT 1 and FAT 2. Then states 73–75 link the old last cluster to the new one in the same way.

13.5 Program walkthrough

The main loop now has three marked parts, *A. prepare*, *B. the one sector access* and *C. evaluate*. Part A decides the sector and the kind of access for every state; for example:

```
264     ; A. prepare = sector number and kind of access for this state
265     ; ====================================================================
266     io:=1                                       ;most states read
267     if (state=0) { lba:=0 }
268     altif (state=1) { lba:=p_lba }
269     altif (state=3) {
270        pm:=c/128
271        lba:=fat_lba+pm
272     }
273     altif (state=5) {
274        pm:=c/128
275        lba:=fat_lba+pm
276     }
277     altif (state=6) { io:=0 }
278     altif (state=7) { lba:=fat_lba+raw32(fi) }
279     altif (state=71) {
280        lba:=fat_lba+raw32(fi)
```

```
281         io:=2
282      }
283      altif (state=72) {
284         lba:=fat_lba+fsz
285         lba:=lba+raw32(fi)
286         io:=2
287      }
288      altif (state=73) {
289         pm:=c/128
290         lba:=fat_lba+pm
291      }
292      altif (state=74) {
293         pm:=c/128
294         lba:=fat_lba+pm
295         io:=2
296      }
297      altif (state=75) {
298         pm:=c/128
299         lba:=fat_lba+pm
300         lba:=lba+fsz
301         io:=2
302      }
303      altif (state=9) { io:=2 }                ;same data sector as state 8
304      altif (state=10) { lba:=dlba }
```

Part C contains the new states. State 7 searches the FAT sector for a free entry and marks it as end of chain:

```
661      altif (state=7) {                        ;search a free FAT entry in sector fi
662         found:=0
663         j:=0
664         loop {
665            newc:=raw32(fi)*128
666            newc:=newc+raw32(j)
667            half:=j/64
668            idx:=j&63
669            idx:=idx*4
670            if (half=0) {
671               c0:=buf_lo[idx]
672               idx:=idx+1
673               c1:=buf_lo[idx]
674               idx:=idx+1
675               c2:=buf_lo[idx]
676               idx:=idx+1
677               c3:=buf_lo[idx]
678            }
679            default {
680               c0:=buf_hi[idx]
681               idx:=idx+1
682               c1:=buf_hi[idx]
683               idx:=idx+1
684               c2:=buf_hi[idx]
685               idx:=idx+1
686               c3:=buf_hi[idx]
687            }
688            c3:=c3&15
689            t:=0                               ;free entry = all 28 bits zero
690            if (c0!=0) { t:=1 }
691            if (c1!=0) { t:=1 }
692            if (c2!=0) { t:=1 }
693            if (c3!=0) { t:=1 }
694            if (newc<2) { t:=1 }               ;clusters 0 and 1 are reserved
695            if (newc>maxc) { t:=1 }            ;beyond the end of the data area
696            if (t=0) {
697               found:=1
698               quit
699            }
700            j:=j+1
701            if (j>=128) { quit }
702         }
703         if (found=1) {                        ;mark newc as end of chain (0FFFFFFF)
704            half:=j/64                         ;idx points to byte 3 of the entry
705            if (half=0) {
706               t:=buf_lo[idx]
707               t:=t&240                        ;keep the 4 reserved bits
708               t:=t+15                         ;(low nibble is 0 now, + acts as OR)
709               buf_lo[idx]:=t
710               idx:=idx-1
711               buf_lo[idx]:=255
712               idx:=idx-1
713               buf_lo[idx]:=255
714               idx:=idx-1
715               buf_lo[idx]:=255
716            }
717            default {
718               t:=buf_hi[idx]
719               t:=t&240
720               t:=t+15
```

```
            buf_hi[idx]:=t
            idx:=idx-1
            buf_hi[idx]:=255
            idx:=idx-1
            buf_hi[idx]:=255
            idx:=idx-1
            buf_hi[idx]:=255
         }
         putc("N") putc("E") putc("W") putc("Space") putc("C") putc("Equal")
         put_hex32(newc)
         put_crlf()
         state:=71
      }
      default {
         fi:=fi+1
         pm:=raw32(fi)
         if (pm>=fsz) {                    ;no free cluster at all
            putc("F") putc("U") putc("L") putc("L")
            put_crlf()
            fail:=1
         }
      }
   }
   altif (state=71) {                      ;FAT 1 written
      if (nf=2) { state:=72 }
      altif (empty=1) { state:=76 }
      default { state:=73 }
   }
   altif (state=72) {                      ;FAT 2 written
      if (empty=1) { state:=76 }
      default { state:=73 }
   }
```

- The candidate cluster number is $fi \cdot 128 + j$ (FAT sector `fi`, entry j).
- An entry is free if all 28 bits are zero; clusters 0 and 1 and numbers beyond `maxc` are skipped.
- The end mark `0FFFFFFF` is written byte by byte; in the highest byte the four reserved bits are kept (`t&240`, then +15).
- States 71 and 72 write the same buffer into FAT 1 and FAT 2.

State 8 places the new bytes behind the old ones:

```
   altif (state=8) {                       ;put the next bytes into the sector
      loop {
         if (nleft=0) { quit }
         if (bo>=512) { quit }
         t:=txt[ci]
         half:=high(bo)
         idx:=low(bo)
         if (half=0) { buf_lo[idx]:=t }
         default { buf_hi[idx]:=t }
         ci:=ci+1
         if (ci>=LEN) { ci:=0 }
         bo:=bo+1
         coff:=coff+1
         fsize:=fsize+1
         nleft:=nleft-1
      }
      state:=9
   }
```

`high(bo)` and `low(bo)` split the 16-bit byte position into the buffer half and the index — the two-halves rule in its shortest form.

13.6 Expected output

With one line (*LINES* = 1, 29 bytes):

```
INIT OK SDSC
FOUND C=00000005 S=00000047
LAST C=00000005 USED=00000047
WRITE LBA=00002098 R=05
WRITE LBA=00002080 R=05
SIZE=00000064 OK
```

The first write is the data sector (8344), the second the root-directory sector (8320) with the new size $71+29=100=$ `0x64`. On the PC, `TEST02.TXT` must now end with the line `MyBug append test PIC18F452`. Every run appends one more line. With *LINES* = 150 (4350 bytes) the first cluster overflows and the allocation states 7 and 71–75 run.

Example 13.1. `TEST02.TXT` *opened on the PC after one run (2026-10-16):*

```
Hello, SD World, here is Luxembourg calling!
Mary had a little lamb.
MyBug append test PIC18F452
```

The two original lines are $46+25=71$ *bytes (each with CR LF); with the new line of 29 bytes the file has* $100=$ `0x64` *bytes, the size the PIC wrote into the directory entry. The PC accepts both the content and the size.*

13.7 Verification

- Compiled with MyBug and tested on the hardware with *LINES* = 1 (2026-10-16): the line is appended, and the PC shows the file with the new line and the correct size. Read-modify-write of a data sector and of a directory sector are thus verified.
- `chkdsk D:` on the PC (read-only, 2026-10-16): "Windows has scanned the file system and found no problems." It also reports 480 752 allocation units of 4096 bytes — exactly the number of clusters computed by hand from the BPB in block B4.
- Cluster allocation (states 7, 71–76), tested with *LINES* = 150 (program `18_sd_8b_append150.jan`, 2026-10-16): the PC shows `TEST02.TXT` with the two original lines followed by the appended lines, without a gap or garbage at the cluster boundary (byte 4096). The text after that boundary can only be read through the new FAT link, so the allocation and the link in the FAT work. The PIC reported the allocation of a new cluster and `SIZE=00001162`; the PC shows the file with 4450 bytes, the old 100 plus $150\cdot 29=4350$. `chkdsk` found no problems.

14 Building block B9: creating a file

Free directory slot

Question: where does a new file get its directory entry?

Concept: a directory has no list of free places; a 32-byte slot is free if its first byte is `E5` (a deleted entry, which may be reused) or `00` (the end mark: this slot and all following ones are unused). Creating a file means filling such a slot with a complete entry: name, attribute, time stamps, first cluster and size.

Example: FAT stores a date as $(year-1980)\cdot 512+month\cdot 32+day$, so 2026-10-16 becomes $46\cdot 512+10\cdot 32+16=23\,888=$ `0x5D50`; a time as $hour\cdot 2048+minute\cdot 32+second/2$, so 12:00:00 becomes `0x6000`.

14.1 What this block adds

Block B9 is block B8 with a different search and a complete entry; allocating the cluster and writing the data use exactly the code of B8, in its "empty file" path:

- **Search (state 2).** Every used entry is compared with the new name `MYBUG01  TXT`; if it exists already, the program stops with `EXISTS`, so a second run cannot create a duplicate. The first free slot (`E5` or `00`) is remembered in `fdlba` (sector) and `fde` (entry number). The search ends at the end mark `00`.
- **A new, empty file (state 20, no card access).** The file starts with size 0 and no cluster; as in B8, *coff* = *cbytes* ("full") forces the allocation of a first cluster, and no link is needed.
- **The complete entry (state 10).** The 32 bytes are built in the array `ent[]` and copied into the free slot of the directory sector (read-modify-write). The time stamps are fixed (2026-10-16, 12:00:00), because the PIC has no real-time clock yet.
- **Order.** Data, then FAT, then the directory entry — the file becomes visible only when it is complete.

Offset	Field	Value written
0–10	name, extension	`MYBUG01  TXT`
11	attribute	20 (archive)
12, 13	reserved, creation time (10 ms)	0, 0
14–17	creation time, date	`6000`, `5D50`
18, 19	last access date	`5D50`
20, 21	first cluster, high word	from `newc`
22–25	write time, date	`6000`, `5D50`
26, 27	first cluster, low word	from `newc`
28–31	size	102 (`0x66`)

Table 5: The directory entry written by block B9 (multi-byte values little-endian).

Nota Bene 14.1. *Two limits are accepted for now: the program writes only a short (8.3) name, no long file name entries — Windows shows the file as* `MYBUG01.TXT`*; and it does not extend the root directory by a new cluster when every slot is in use (it stops with* `DIR FULL`*).*

14.2 Program walkthrough

State 20 turns the search result into a new, empty file; it accesses no sector and only prepares the variables, so that the allocation code of B8 takes over:

```
815     ; state 20 (internal) = start a new, empty file in the free slot
816     if (state=20) {
817        dlba:=fdlba
818        de:=fde
819        c:=0
820        fsize:=0
821        empty:=1
822        coff:=cbytes                          ;behaves like a full cluster -> allocate
823        putc("F") putc("R") putc("E") putc("E") putc("Space") putc("L") putc("B") putc("A")
          putc("Equal")
824        put_hex32(dlba)
825        putc("Space") putc("E") putc("Equal")
826        put_hex(de)
827        put_crlf()
828        state:=6
829     }
```

The complete entry is assembled in `ent[]` with constant indices — a direct image of Table 5 — and then copied into the free slot:

```
753     altif (state=10) {                       ;build the complete 32-byte entry in ent[]
754        take j from [0,11) {                  ;8.3 name without the dot
755           t:=target[j]
756           ent[j]:=t
757        }
758        ent[11]:=32                           ;attribute 20 = archive (normal file)
759        ent[12]:=0                            ;reserved (lower-case flags)
760        ent[13]:=0                            ;creation time, 10 ms units
761        ent[14]:=0                            ;creation time 12h00m00s = 6000
762        ent[15]:=96
763        ent[16]:=80                           ;creation date 2026-10-16 = 5D50
764        ent[17]:=93
765        ent[18]:=80                           ;last access date
766        ent[19]:=93
767        ent[22]:=0                            ;write time
768        ent[23]:=96
769        ent[24]:=80                           ;write date
770        ent[25]:=93
771        pm:=firstnew                          ;first cluster, low word 26/27, high 20/21
772        t:=raw8(pm)
773        ent[26]:=t
774        pm:=pm/256
775        t:=raw8(pm)
776        ent[27]:=t
777        pm:=pm/256
778        t:=raw8(pm)
779        ent[20]:=t
780        pm:=pm/256
781        t:=raw8(pm)
782        ent[21]:=t
783        pm:=fsize                             ;size, offset 28..31
784        t:=raw8(pm)
785        ent[28]:=t
786        pm:=pm/256
```

```
787         t:=raw8(pm)
788         ent[29]:=t
789         pm:=pm/256
790         t:=raw8(pm)
791         ent[30]:=t
792         pm:=pm/256
793         t:=raw8(pm)
794         ent[31]:=t
795         half:=de/8                          ;copy ent[] into the free slot de
796         base:=de%8
797         base:=base*32
798         take j from [0,32) {
799            idx:=base+j
800            t:=ent[j]
801            if (half=0) { buf_lo[idx]:=t }
802            default { buf_hi[idx]:=t }
803         }
804         state:=11
805      }
```

14.3 Expected output

```
INIT OK SDSC
FREE LBA=00002080 E=xx
NEW C=0000xxxx
WRITE LBA=00000328 R=05
WRITE LBA=000011D4 R=05
WRITE LBA=0000xxxx R=05
WRITE LBA=00002080 R=05
SIZE=00000066 OK
```

The free slot lies in the first root sector if a deleted or the end entry is there. Two FAT writes (FAT 1 and FAT 2), one data sector, one directory sector. On the PC, `MYBUG01.TXT` must appear with 102 bytes, three times the line "Created by MyBug on a PIC18F452." and the date 16.10.2026; `chkdsk` must find no problems. A second run reports `EXISTS`.

Example 14.1. *Output on the author's card (2026-10-16):*

```
INIT OK SDSC
FREE LBA=00002082 E=01
NEW C=00000052
WRITE LBA=00000328 R=05
WRITE LBA=000011D4 R=05
WRITE LBA=00002300 R=05
WRITE LBA=00002082 R=05
SIZE=00000066 OK
```

The first free slot is entry 1 of the third *root sector (LBA* $8320+2$*): the short entries of B5 together with the long-name entries of the Windows files fill more than two sectors. The new cluster* `0x52` *is the one right behind the last file of B5 (*`DATA_A~5.TXT` *at* `0x51`*); its first sector is* $8320+(\texttt{0x52}-2)\cdot 8 = 8960 = \texttt{0x2300}$*, exactly the LBA of the data write.*

14.4 Verification

- Compiled with MyBug and tested on the hardware (2026-10-16): free slot found, first cluster allocated in both FATs, data and directory entry written, size 102 bytes; all sector numbers agree with the hand calculation.
- A second run stops with `EXISTS`: the program finds its own entry again by name, so the duplicate check works — and the new entry is readable by the PIC's own directory search.
- On the PC, `MYBUG01.TXT` opens with exactly the three lines "Created by MyBug on a PIC18F452." — a file created entirely by the PIC is accepted by Windows. The Explorer shows the date written by the PIC (16.10.2026), and a final `chkdsk` finds no errors.

15 Conclusion

15.1 What has been achieved

A PIC18F452 with 1536 B of RAM can read and write a FAT32 card that a PC accepts without complaint. Table 6 summarizes the nine building blocks; every one of them was run on the hardware, and every computed number was checked by hand.

Block	Program	Verified on 2026-10-16
B1	`18_sd_1_spi`	SPI loopback at 250 kHz, 1 MHz and 4 MHz without error
B2	`18_sd_2_init`	card initialized (CMD0, CMD8, ACMD41, CMD58), card swap in operation
B3	`18_sd_3_read`	sector 0 read, boot signature `55 AA` found
B4	`18_sd_4_bpb`	partition and BPB read, `fat_lba` and `data_lba` equal to the hand calculation
B5	`18_sd_5_dir`	root directory listed, sizes and clusters consistent
B6	`18_sd_6_read_file`	127 134-byte file read across 32 clusters (first run of the FAT chain)
B7	`18_sd_7_write`	sector written, read back without error and restored
B8	`18_sd_8_append`	line appended; with 150 lines a new cluster allocated and linked in both FATs
B9	`18_sd_9_create`	new file created; content, size and date correct on the PC, `chkdsk` clean

Table 6: The nine building blocks and their verification.

15.2 What there is to learn

- **Layers make a complex system manageable.** Byte, command, sector, volume, cluster and file each have their own small set of rules (Figure 7); none of the programs is difficult once the layer below it works.
- **Small steps find errors early.** Because every block added only one capability, each problem appeared in the few new lines of one program, with a UART output that showed exactly where it occurred.
- **A specification explains observations.** The byte-address "mystery" of the author's earlier experiments (Baumann, 2024) turned out to be the documented difference between SDSC and SDHC cards, visible in a single bit of the OCR register (Section 2.6).
- **Order protects data.** Writing data first, then the FAT and the directory entry last keeps the existing files intact even if the power fails in the middle of an operation.
- **Check against an independent reference.** Hand calculations, the PC's view of the card and `chkdsk` confirmed every step; the PIC's own output was never the only proof.

15.3 What MyBug gained

The project was also a test of the PIC18F452 back end of MyBug. New are the SPI snippets `.configure_spi`, `.spi_set_speed` and `.spi_xfer`. Proven on the hardware for the first time were 32-bit arithmetic (assembling, multiplying by a constant, adding, dividing by 256, narrowing with `raw8()`), two arrays of 256 bytes, constant indices into arrays outside bank 0, and functions with six parameters. One trap was found and documented: an identifier must not begin with `inf` or `nan`.

16 Outlook

The programs cover what a simple data logger needs. The following extensions are natural next building blocks, roughly in order of usefulness:

1. **An SDHC card.** All tests used a 2 GB SDSC card. The SDHC path (block addressing, HCS bit in ACMD41) is implemented but has not yet run on the hardware.
2. **The permanent hardware** with the 74HCT125 on MISO (Figure 8), followed by a long-term test.
3. **A library instead of copies.** The helpers (`xchg`, `sd_cmd`, `sd_init`, `mk32`, ...) are copied into every program; a MyBug library file would remove the repetition. This needs an answer to the open question whether subroutines can reach main's arrays.

4. **512-byte arrays.** Lifting MyBug's 256-byte limit for the PIC18F452 (ISS-004) would remove the two buffer halves.
5. **Real time stamps** from a real-time clock instead of the fixed date.
6. **More of FAT32:** updating the FSInfo free-cluster count, extending a full directory by a new cluster, subdirectories, and long file names.
7. **Speed:** multi-block commands (CMD18, CMD25) and fewer status lines on the UART.
8. **The application:** the data logger for the author's home heating system, for which this project was started.

License, trademarks and acknowledgments

License. © 2026 Claude Baumann. This report and the test programs `18_sd_*.jan` are licensed under the Creative Commons license *Attribution-NonCommercial-ShareAlike 4.0 International* (CC BY-NC-SA 4.0, `https://creativecommons.org/licenses/by-nc-sa/4.0/`). You may copy, distribute and adapt them under three conditions: you must name the author and the license (*attribution*), you may not use them for commercial purposes (*non-commercial*), and adapted versions must be distributed under the same license (*share alike*). The MyBug compiler and its libraries are not covered by this license.

Acknowledgments. The test programs, the SPI extensions of the MyBug library and this report were developed with the help of an AI assistant (Claude, Anthropic). The author reviewed every program, compiled it with MyBug, ran it on the hardware and checked the results; every reference was checked against its source.

Trademarks. The SD, SDHC, miniSDHC, microSDHC, SDXC and microSDXC logos are trademarks of SD-3C LLC (wording as recommended by the SD Association). Microchip, PIC and MPLAB are registered trademarks of Microchip Technology Inc. Microsoft and Windows are trademarks of the Microsoft group of companies. Raspberry Pi is a trademark of Raspberry Pi Ltd. AZ-Delivery is the vendor of the micro-SD module used. All other names are the property of their respective owners.

This educational project relies only on publicly available documents: the *simplified* version of the SD physical layer specification (SD Card Association, 2023), the FAT32 specification (Microsoft Corporation, 2000) and the Microchip data sheet (Microchip Technology Inc., 2006). No code from these documents or from third-party libraries is used; all programs were written for this project in MyBug. Readers who plan a commercial product with an SD card interface should check the licensing conditions of the SD Association themselves.

References

Baumann, C. (2024). *Communication with SD cards* [Lab report, version 1.2]. Computarium.

Ibrahim, D. (2010). *SD card projects using the PIC microcontroller*. Newnes.

Layton, J. (2018). *Using the SPI module on 8-bit PIC microcontrollers* (Technical Brief TB3192, DS90003192A). Microchip Technology Inc.

Microchip Technology Inc. (2006). *PIC18FXX2 data sheet: High-performance, enhanced flash microcontrollers with 10-bit A/D* (DS39564C).

Microsoft Corporation. (2000). *Microsoft Extensible Firmware Initiative FAT32 file system specification* (Version 1.03).

onsemi. (2025). *74VHC125: Quad buffer with 3-state outputs* (Rev. 3) [Data sheet]. Semiconductor Components Industries, LLC.

Reen, P., & Mohanswamy, N. (2008). *Implementing file I/O functions using Microchip's memory disk drive file system library* (Application Note AN1045, DS01045B). Microchip Technology Inc.

SD Card Association, Technical Committee. (2023). *SD specifications part 1: Physical layer simplified specification* (Version 9.10).

A Listings

A.1 Test program `18_sd_1_spi.jan`

```
; Program name: 18_sd_1_spi.jan
; Object file name: 18_sd_1_spi.asm
; Uses: ascii.jan
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 1 (SPI bring-up, no card yet)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI), RC5 (SDO), RC2 (CS, held high),
;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
; Hardware: LOOPBACK jumper RC5 (SDO) -> RC4 (SDI). No SD card connected.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 15/10/2026
; Version: 0.1
; Description: tests the MSSP SPI master snippets .configure_spi, .spi_set_speed and
;              .spi_xfer. For each of the three clocks (SPI_SLOW 250 kHz, SPI_MEDIUM
;              1 MHz, SPI_FAST 4 MHz) the bytes 0..254 are sent; with the loopback
;              jumper every received byte must equal the sent byte. One line per
;              clock is sent via UART, e.g. "S0 E=00 L=FE" (speed 0, error count in
;              hex, last received byte in hex). Green LED = all three passes without
;              error, red LED = at least one error. Remove the jumper -> SDI floats
;              (or reads 00/FF) and errors must appear: proves the test is real.
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low), idle high

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;deselect the (future) card before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

raw8 pass,tx_val,rx_val,errs,fails,digit,c
loop {
   fails:=0
   take pass from [0,3) {
      if (pass=0) { .spi_set_speed(SPI_SLOW) }
      altif (pass=1) { .spi_set_speed(SPI_MEDIUM) }
      default { .spi_set_speed(SPI_FAST) }

      errs:=0
      take tx_val from [0,255) {
         .spi_xfer(tx_val,rx_val,context,local_id)
         if (rx_val=tx_val) { }
         default { errs:=errs+1 }
      }
      fails:=fails+errs

      ;report line  "S<pass> E=<errs hex> L=<last rx hex>" CR LF
      c:="S"
      .uart_tx_byte(c,context,local_id)
      c:=digit_to_ascii(pass)
      .uart_tx_byte(c,context,local_id)
      c:="Space"
      .uart_tx_byte(c,context,local_id)
      c:="E"
      .uart_tx_byte(c,context,local_id)
      c:="Equal"
      .uart_tx_byte(c,context,local_id)
      digit:=errs/16
      c:=digit_to_ascii(digit)
      .uart_tx_byte(c,context,local_id)
      digit:=errs%16
      c:=digit_to_ascii(digit)
      .uart_tx_byte(c,context,local_id)
      c:="Space"
      .uart_tx_byte(c,context,local_id)
      c:="L"
      .uart_tx_byte(c,context,local_id)
      c:="Equal"
      .uart_tx_byte(c,context,local_id)
      digit:=rx_val/16
```

```
      c:=digit_to_ascii(digit)
      .uart_tx_byte(c,context,local_id)
      digit:=rx_val%16
      c:=digit_to_ascii(digit)
      .uart_tx_byte(c,context,local_id)
      c:="CR"
      .uart_tx_byte(c,context,local_id)
      c:="LF"
      .uart_tx_byte(c,context,local_id)
   }

   if (fails=0) {
      setbit(LEDPORT,GREEN)
      clearbit(LEDPORT,RED)
   }
   default {
      clearbit(LEDPORT,GREEN)
      setbit(LEDPORT,RED)
   }
   delay(1000)
}
```

A.2 Test program `18_sd_2_init.jan`

```
; Program name: 18_sd_2_init.jan
; Object file name: 18_sd_2_init.asm
; Uses: ascii.jan
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 2 (card initialization)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card inserted.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 16/10/2026
; Version: 0.1
; Description: brings an SD card from power-up into SPI mode and reports every answer
;              via UART (hex), then repeats every 2 s (card may be swapped meanwhile).
;   1. CS high, 80 clocks (>= 74 required) at SPI_SLOW (250 kHz)
;   2. CMD0  (GO_IDLE_STATE)         expected R1 = 01 (idle)
;   3. CMD8  (SEND_IF_COND, 0x1AA)   expected R1 = 01, R7 = 000001AA (card V2)
;                                    R1 = 05 (illegal command) means an old V1 card
;   4. CMD55 + ACMD41 (HCS for V2)   repeated until R1 = 00 (initialization done)
;   5. CMD58 (READ_OCR)              OCR bit 30 (CCS) set = SDHC/SDXC, block addressing
;   then SPI_FAST (4 MHz) for the following blocks.
; Expected output (SDHC card), one block every 2 s
;   CMD0 R1=01
;   CMD8 R1=01 R7=000001AA
;   ACMD41 R1=00 N=xx          (N = number of repetitions, card dependent)
;   CMD58 R1=00 OCR=C0FF8000   (first byte C0 = powered up + CCS)
;   SDHC OK
; Green LED = initialization succeeded, red LED = it failed.
; Measure here the MISO idle-high level at RC4 (spec VIH >= 0.7 VDD = 3.5 V).
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;card deselected before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ------------------------------------------------------------------------
; helpers
; ------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
   return(r)
}

```

```
; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
; (CRC only checked by the card for CMD0 and CMD8, sent correctly for all anyway)
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                     ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                           ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }           ;no answer -> r stays FF
   }
   return(r)
}

; CS high, then 8 more clocks so the card releases MISO
subroutine sd_deselect() {
   raw8 r
   setbit(CSPORT,CS)
   r:=xchg(255)
}

subroutine putc(raw8 ch) {
   .uart_tx_byte(ch,context,local_id)
}

subroutine put_hex(raw8 v) {
   raw8 d,c
   d:=v/16
   c:=digit_to_ascii(d)
   putc(c)
   d:=v%16
   c:=digit_to_ascii(d)
   putc(c)
}

subroutine put_crlf() {
   putc("CR")
   putc("LF")
}

; the " R1=xx" part of every report line (text is sent letter by letter)
subroutine put_r1(raw8 v) {
   putc("Space")
   putc("R")
   putc("_onE")
   putc("Equal")
   put_hex(v)
}

; ------------------------------------------------------------------------
; main
; ------------------------------------------------------------------------
raw8 r,k,tries,fail,v2,hcs,crc41,t
array raw8 resp[4]                  ;the 4 bytes following R1 (R7 of CMD8, OCR of CMD58)

loop {
   fail:=0
   clearbit(LEDPORT,GREEN)
   clearbit(LEDPORT,RED)

   ; --- 1. power-up clocks, CS high, 10 x FF = 80 clocks at 250 kHz -------
   setbit(CSPORT,CS)
   .spi_set_speed(SPI_SLOW)
   take k from [0,10) { r:=xchg(255) }

   ; --- 2. CMD0 = software reset, enters SPI mode (CS low!) ----------------
   clearbit(CSPORT,CS)
   r:=sd_cmd(64,0,0,0,0,149)
   sd_deselect()
   putc("C") putc("M") putc("D") putc("_zerO")
   put_r1(r)
   put_crlf()
   if (r!=1) { fail:=1 }

   ; --- 3. CMD8 = interface condition, 2.7-3.6 V, check pattern AA ---------
   clearbit(CSPORT,CS)
   r:=sd_cmd(72,0,0,1,170,135)
   take k from [0,4) {
      t:=xchg(255)
```

```
148         resp[k]:=t
149      }
150      sd_deselect()
151      putc("C") putc("M") putc("D") putc("_eighT")
152      put_r1(r)
153      v2:=0
154      if (r=1) {                        ;V2 card, the 4 R7 bytes are valid
155         putc("Space") putc("R") putc("_seveN") putc("Equal")
156         take k from [0,4) {
157            t:=resp[k]
158            put_hex(t)
159         }
160         t:=resp[3]
161         if (t=170) { v2:=1 }           ;echo of the check pattern
162         default { fail:=1 }
163      }
164      altif (r=5) { }                   ;illegal command = V1 card (SDSC), no R7
165      default { fail:=1 }
166      put_crlf()
167
168      ; --- 4. ACMD41 = start initialization, repeat until R1 = 00 -------------
169      if (v2=1) { hcs:=64  crc41:=119 }   ;HCS bit 30 = host supports SDHC
170      default { hcs:=0  crc41:=229 }
171      tries:=0
172      loop {
173         clearbit(CSPORT,CS)
174         r:=sd_cmd(119,0,0,0,0,101)       ;CMD55 = next command is application specific
175         r:=sd_cmd(105,hcs,0,0,0,crc41)   ;ACMD41
176         sd_deselect()
177         if (r=0) { quit }
178         tries:=tries+1
179         if (tries>=200) { quit }         ;200 x 10 ms = 2 s (spec 1 s)
180         delay(10)
181      }
182      putc("A") putc("C") putc("M") putc("D") putc("_fouR") putc("_onE")
183      put_r1(r)
184      putc("Space") putc("N") putc("Equal")
185      put_hex(tries)
186      put_crlf()
187      if (r!=0) { fail:=1 }
188
189      ; --- 5. CMD58 = read OCR, bit 30 (CCS) tells SDHC from SDSC -------------
190      clearbit(CSPORT,CS)
191      r:=sd_cmd(122,0,0,0,0,253)
192      take k from [0,4) {
193         t:=xchg(255)
194         resp[k]:=t
195      }
196      sd_deselect()
197      putc("C") putc("M") putc("D") putc("_fivE") putc("_eighT")
198      put_r1(r)
199      putc("Space") putc("O") putc("C") putc("R") putc("Equal")
200      take k from [0,4) {
201         t:=resp[k]
202         put_hex(t)
203      }
204      put_crlf()
205      if (r!=0) { fail:=1 }
206
207      ; --- result -------------------------------------------------------------
208      t:=resp[0]
209      t:=t&64                           ;CCS = OCR bit 30 = bit 6 of the first byte
210      if (t!=0) { putc("S") putc("D") putc("H") putc("C") }
211      default { putc("S") putc("D") putc("S") putc("C") }
212      putc("Space")
213      if (fail=0) {
214         putc("O") putc("K")
215         setbit(LEDPORT,GREEN)
216         .spi_set_speed(SPI_FAST)      ;card ready, data transfers may run at 4 MHz
217      }
218      default {
219         putc("F") putc("A") putc("I") putc("L")
220         setbit(LEDPORT,RED)
221      }
222      put_crlf()
223      put_crlf()
224      delay(2000)
225   }
```

A.3 Test program `18_sd_3_read.jan`

```
1  ; Program name: 18_sd_3_read.jan
2  ; Object file name: 18_sd_3_read.asm
3  ; Uses: ascii.jan
```

```
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 3 (read one sector)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card inserted.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 16/10/2026
; Version: 0.1
; Description: initializes the card (block B2, now silent in function sd_init), reads
;              sector 0 with CMD17 and sends it as a hex dump via UART, then checks the
;              boot signature 55 AA at offset 510/511. Repeats every 5 s.
;   - the 512 data bytes go into TWO arrays of 256 bytes (buf_lo = bytes 0..255,
;     buf_hi = bytes 256..511), because 18F452 arrays are kept <= 256 bytes (ISS-004)
;   - a raw8 index runs 0..255 and wraps to 0 -> "quit when k = 0" ends a 256-byte loop
;   - CMD17 argument = sector number (SDHC) or byte address (SDSC); for sector 0 both
;     are 0, so this block works for both card types without a distinction yet
; Expected output
;   INIT OK SDHC
;   CMD17 R1=00 TOK=FE
;   000 33 C0 8E D0 ...        (32 lines of 16 bytes, offset 000..1F0 in hex)
;   ...
;   1F0 00 00 ... 00 55 AA
;   SIG 55AA OK
; Green LED = sector read and signature found, red LED = error.
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;card deselected before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ------------------------------------------------------------------------
; helpers (from block B2)
; ------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
   return(r)
}

; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                     ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                           ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }           ;no answer -> r stays FF
   }
   return(r)
}

; CS high, then 8 more clocks so the card releases MISO
subroutine sd_deselect() {
   raw8 r
   setbit(CSPORT,CS)
   r:=xchg(255)
}

subroutine putc(raw8 ch) {
   .uart_tx_byte(ch,context,local_id)
}
```

```
91
92  subroutine put_hex(raw8 v) {
93     raw8 d,c
94     d:=v/16
95     c:=digit_to_ascii(d)
96     putc(c)
97     d:=v%16
98     c:=digit_to_ascii(d)
99     putc(c)
100 }
101
102 subroutine put_crlf() {
103    putc("CR")
104    putc("LF")
105 }
106
107 ; ------------------------------------------------------------------------
108 ; block B2 as one silent function.
109 ; returns 0 = initialization failed, 1 = SDSC (byte addresses), 2 = SDHC (block addresses)
110 ; st = 0 as long as every step succeeded; each later step runs only if st = 0
111 ; ------------------------------------------------------------------------
112 function raw8 sd_init() {
113    raw8 r,k,st,tries,v2,hcs,crc41,last,ocr0,kind
114    st:=0
115    kind:=0
116    setbit(CSPORT,CS)
117    .spi_set_speed(SPI_SLOW)
118    take k from [0,10) { r:=xchg(255) }       ;80 clocks with CS high
119
120    clearbit(CSPORT,CS)
121    r:=sd_cmd(64,0,0,0,0,149)                 ;CMD0 -> expect 01
122    sd_deselect()
123    if (r!=1) { st:=1 }
124
125    v2:=0
126    if (st=0) {
127       clearbit(CSPORT,CS)
128       r:=sd_cmd(72,0,0,1,170,135)            ;CMD8 -> expect 01 + 000001AA
129       take k from [0,4) { last:=xchg(255) } ;keep only the echo byte AA
130       sd_deselect()
131       if (r=1) {
132          if (last=170) { v2:=1 }
133          default { st:=1 }
134       }
135       altif (r=5) { }                        ;V1 card, no R7
136       default { st:=1 }
137    }
138
139    if (st=0) {
140       if (v2=1) { hcs:=64  crc41:=119 }
141       default { hcs:=0  crc41:=229 }
142       tries:=0
143       loop {
144          clearbit(CSPORT,CS)
145          r:=sd_cmd(119,0,0,0,0,101)          ;CMD55
146          r:=sd_cmd(105,hcs,0,0,0,crc41)      ;ACMD41
147          sd_deselect()
148          if (r=0) { quit }
149          tries:=tries+1
150          if (tries>=200) { quit }
151          delay(10)
152       }
153       if (r!=0) { st:=1 }
154    }
155
156    if (st=0) {
157       clearbit(CSPORT,CS)
158       r:=sd_cmd(122,0,0,0,0,253)             ;CMD58 -> OCR
159       ocr0:=xchg(255)                        ;first OCR byte holds CCS (bit 6)
160       take k from [0,3) { last:=xchg(255) }
161       sd_deselect()
162       if (r!=0) { st:=1 }
163    }
164
165    if (st=0) {
166       ocr0:=ocr0&64
167       if (ocr0!=0) { kind:=2 }
168       default { kind:=1 }
169       .spi_set_speed(SPI_FAST)               ;4 MHz for the data transfer
170    }
171    return(kind)
172 }
173
174 ; ------------------------------------------------------------------------
175 ; main
176 ; ------------------------------------------------------------------------
177 raw8 r,k,t,mrk,fail,kind,half,c,s1,s2
```

```
raw16 n
array raw8 buf_lo[256]              ;sector bytes   0..255
array raw8 buf_hi[256]              ;sector bytes 256..511

loop {
   fail:=0
   clearbit(LEDPORT,GREEN)
   clearbit(LEDPORT,RED)

   ; --- initialization (block B2) ----------------------------------------
   kind:=sd_init()
   putc("I") putc("N") putc("I") putc("T") putc("Space")
   if (kind=0) {
      putc("F") putc("A") putc("I") putc("L")
      fail:=1
   }
   altif (kind=1) { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("S") putc("C")
      }
   default { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("H") putc("C") }
   put_crlf()

   ; --- CMD17 = READ_SINGLE_BLOCK, sector 0 ------------------------------
   mrk:=255
   if (fail=0) {
      clearbit(CSPORT,CS)
      r:=sd_cmd(81,0,0,0,0,85)              ;CMD17, argument 0, CRC 0x55
      if (r=0) {
         n:=0                               ;wait for the data token FE
         loop {                             ;(card may need up to ~100 ms)
            mrk:=xchg(255)
            if (mrk!=255) { quit }
            n:=n+1
            if (n>=50000) { quit }
         }
      }
      if (mrk=254) {
         k:=0                               ;bytes 0..255
         loop {
            t:=xchg(255)
            buf_lo[k]:=t
            k:=k+1
            if (k=0) { quit }               ;raw8 wrapped after 256 bytes
         }
         loop {                             ;bytes 256..511 (k is 0 again)
            t:=xchg(255)
            buf_hi[k]:=t
            k:=k+1
            if (k=0) { quit }
         }
         t:=xchg(255)                       ;CRC16 of the data block, ignored
         t:=xchg(255)
      }
      sd_deselect()
      putc("C") putc("M") putc("D") putc("_onE") putc("_seveN")
      putc("Space") putc("R") putc("_onE") putc("Equal")
      put_hex(r)
      putc("Space") putc("T") putc("O") putc("K") putc("Equal")
      put_hex(mrk)
      put_crlf()
      if (mrk!=254) { fail:=1 }
   }

   ; --- hex dump, 32 lines of 16 bytes -----------------------------------
   if (fail=0) {
      take half from [0,2) {
         k:=0
         loop {
            t:=k&15
            if (t=0) {                      ;line start, offset half*256 + k
               c:=digit_to_ascii(half)
               putc(c)
               put_hex(k)
               putc("Space")
            }
            if (half=0) { t:=buf_lo[k] }
            default { t:=buf_hi[k] }
            put_hex(t)
            putc("Space")
            k:=k+1
            t:=k&15
            if (t=0) { put_crlf() }
            if (k=0) { quit }
         }
      }

      ; --- boot signature, offsets 510 and 511 = buf_hi[254], buf_hi[255] --
      s1:=buf_hi[254]
```

```
        s2:=buf_hi[255]
        putc("S") putc("I") putc("G") putc("Space")
        put_hex(s1)
        put_hex(s2)
        putc("Space")
        if (s1=85) {
           if (s2=170) { putc("O") putc("K") }
           default { fail:=1 }
        }
        default { fail:=1 }
        if (fail!=0) { putc("F") putc("A") putc("I") putc("L") }
        put_crlf()
    }

    if (fail=0) { setbit(LEDPORT,GREEN) }
    default { setbit(LEDPORT,RED) }
    put_crlf()
    delay(5000)
}
```

A.4 Test program `18_sd_4_bpb.jan`

```
; Program name: 18_sd_4_bpb.jan
; Object file name: 18_sd_4_bpb.asm
; Uses: ascii.jan
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 4 (partition table + boot sector)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card (FAT32, MBR) inserted.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 16/10/2026
; Version: 0.1
; Description: reads two sectors with the same code (take step from [0,2))
;   step 0 = sector 0 (MBR) -> partition entry 1 at offset 446
;            type byte (offset 450) must be 0B or 0C (FAT32),
;            start LBA = 4 bytes little-endian at offset 454..457
;   step 1 = sector p_lba (FAT32 boot sector) -> BIOS parameter block (BPB)
;            BytsPerSec (11/12), SecPerClus (13), RsvdSecCnt (14/15), NumFATs (16),
;            FATSz32 (36..39), RootClus (44..47), "FAT32" at offset 82
;   and computes
;            fat_lba  = p_lba + RsvdSecCnt                 (first sector of FAT 1)
;            data_lba = fat_lba + NumFATs * FATSz32         (first sector of cluster 2)
;   The sector number is turned into the CMD17 argument by the card type
;            SDHC  argument = LBA            (block address)
;            SDSC  argument = LBA * 512      (byte address)
;   All multi-byte values are printed in hex, most significant byte first.
; Expected output (values depend on the card; the author's card has p_lba = 128)
;   INIT OK SDSC
;   MBR SIG OK TYPE=0C LBA=00000080
;   BPB SIG OK FAT32
;   BPS=0200 SPC=xx RSV=xxxx NF=02
;   FSZ=xxxxxxxx ROOT=00000002
;   FAT=xxxxxxxx DATA=xxxxxxxx
; Green LED = all checks passed, red LED = error. Repeats every 5 s.
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;card deselected before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ---------------------------------------------------------------------------
; helpers (from blocks B2/B3)
; ---------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
```

```
   return(r)
}

; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                      ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                            ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }            ;no answer -> r stays FF
   }
   return(r)
}

; CS high, then 8 more clocks so the card releases MISO
subroutine sd_deselect() {
   raw8 r
   setbit(CSPORT,CS)
   r:=xchg(255)
}

subroutine putc(raw8 ch) {
   .uart_tx_byte(ch,context,local_id)
}

subroutine put_hex(raw8 v) {
   raw8 d,c
   d:=v/16
   c:=digit_to_ascii(d)
   putc(c)
   d:=v%16
   c:=digit_to_ascii(d)
   putc(c)
}

subroutine put_crlf() {
   putc("CR")
   putc("LF")
}

; (new) four bytes (most significant first) -> one raw32 value
function raw32 mk32(raw8 b3,raw8 b2,raw8 b1,raw8 b0) {
   raw32 v
   v:=raw32(b3)
   v:=v*256
   v:=v+raw32(b2)
   v:=v*256
   v:=v+raw32(b1)
   v:=v*256
   v:=v+raw32(b0)
   return(v)
}

; (new) raw32 -> 8 hex digits; raw8(w) keeps the low byte (narrowing), w/256 drops it
subroutine put_hex32(raw32 v) {
   raw32 w
   raw8 b3,b2,b1,b0
   w:=v
   b0:=raw8(w)
   w:=w/256
   b1:=raw8(w)
   w:=w/256
   b2:=raw8(w)
   w:=w/256
   b3:=raw8(w)
   put_hex(b3)
   put_hex(b2)
   put_hex(b1)
   put_hex(b0)
}

; ------------------------------------------------------------------------
; block B2 as one silent function (unchanged from B3).
; returns 0 = initialization failed, 1 = SDSC (byte addresses), 2 = SDHC (block addresses)
; ------------------------------------------------------------------------
function raw8 sd_init() {
   raw8 r,k,st,tries,v2,hcs,crc41,last,ocr0,kind
```

```
    st:=0
    kind:=0
    setbit(CSPORT,CS)
    .spi_set_speed(SPI_SLOW)
    take k from [0,10) { r:=xchg(255) }       ;80 clocks with CS high

    clearbit(CSPORT,CS)
    r:=sd_cmd(64,0,0,0,0,149)                 ;CMD0 -> expect 01
    sd_deselect()
    if (r!=1) { st:=1 }

    v2:=0
    if (st=0) {
       clearbit(CSPORT,CS)
       r:=sd_cmd(72,0,0,1,170,135)            ;CMD8 -> expect 01 + 000001AA
       take k from [0,4) { last:=xchg(255) } ;keep only the echo byte AA
       sd_deselect()
       if (r=1) {
          if (last=170) { v2:=1 }
          default { st:=1 }
       }
       altif (r=5) { }                        ;V1 card, no R7
       default { st:=1 }
    }

    if (st=0) {
       if (v2=1) { hcs:=64  crc41:=119 }
       default { hcs:=0  crc41:=229 }
       tries:=0
       loop {
          clearbit(CSPORT,CS)
          r:=sd_cmd(119,0,0,0,0,101)          ;CMD55
          r:=sd_cmd(105,hcs,0,0,0,crc41)      ;ACMD41
          sd_deselect()
          if (r=0) { quit }
          tries:=tries+1
          if (tries>=200) { quit }
          delay(10)
       }
       if (r!=0) { st:=1 }
    }

    if (st=0) {
       clearbit(CSPORT,CS)
       r:=sd_cmd(122,0,0,0,0,253)             ;CMD58 -> OCR
       ocr0:=xchg(255)                        ;first OCR byte holds CCS (bit 6)
       take k from [0,3) { last:=xchg(255) }
       sd_deselect()
       if (r!=0) { st:=1 }
    }

    if (st=0) {
       ocr0:=ocr0&64
       if (ocr0!=0) { kind:=2 }
       default { kind:=1 }
       .spi_set_speed(SPI_FAST)               ;4 MHz for the data transfer
    }
    return(kind)
}

; ------------------------------------------------------------------------
; main
; ------------------------------------------------------------------------
raw8 r,k,t,mrk,fail,kind,step,s1,s2,ptype,nf,i,a3,a2,a1,a0,c3,c2,c1,c0
raw16 n
raw32 lba,adr,p_lba,rsvd,fsz,fat_lba,data_lba
array raw8 buf_lo[256]              ;sector bytes   0..255
array raw8 buf_hi[256]              ;sector bytes 256..511

loop {
   fail:=0
   clearbit(LEDPORT,GREEN)
   clearbit(LEDPORT,RED)

   kind:=sd_init()
   putc("I") putc("N") putc("I") putc("T") putc("Space")
   if (kind=0) {
      putc("F") putc("A") putc("I") putc("L")
      fail:=1
   }
   altif (kind=1) { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("S") putc("C")
      }
   default { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("H") putc("C") }
   put_crlf()

   lba:=0                                   ;step 0 reads sector 0 (MBR)
   take step from [0,2) {
```

```
236         if (fail=0) {
237            ; --- sector number -> CMD17 argument (4 bytes, MSB first) --------
238            adr:=lba
239            if (kind=1) { adr:=lba*512 }       ;SDSC = byte address
240            a0:=raw8(adr)
241            adr:=adr/256
242            a1:=raw8(adr)
243            adr:=adr/256
244            a2:=raw8(adr)
245            adr:=adr/256
246            a3:=raw8(adr)
247
248            ; --- CMD17 + data token + 512 bytes (as in block B3) -------------
249            mrk:=255
250            clearbit(CSPORT,CS)
251            r:=sd_cmd(81,a3,a2,a1,a0,255)      ;CRC not checked for CMD17
252            if (r=0) {
253               n:=0
254               loop {
255                  mrk:=xchg(255)
256                  if (mrk!=255) { quit }
257                  n:=n+1
258                  if (n>=50000) { quit }
259               }
260            }
261            if (mrk=254) {
262               k:=0
263               loop {
264                  t:=xchg(255)
265                  buf_lo[k]:=t
266                  k:=k+1
267                  if (k=0) { quit }
268               }
269               loop {
270                  t:=xchg(255)
271                  buf_hi[k]:=t
272                  k:=k+1
273                  if (k=0) { quit }
274               }
275               t:=xchg(255)                    ;CRC16, ignored
276               t:=xchg(255)
277            }
278            sd_deselect()
279            if (mrk!=254) {
280               fail:=1
281               putc("R") putc("E") putc("A") putc("D") putc("Space")
282               putc("F") putc("A") putc("I") putc("L")
283            }
284         }
285
286         if (fail=0) {
287            if (step=0) { putc("M") putc("B") putc("R") }
288            default { putc("B") putc("P") putc("B") }
289            putc("Space") putc("S") putc("I") putc("G") putc("Space")
290            s1:=buf_hi[254]
291            s2:=buf_hi[255]
292            if (s1=85) {
293               if (s2=170) { putc("O") putc("K") }
294               default { fail:=1 }
295            }
296            default { fail:=1 }
297            if (fail!=0) { putc("F") putc("A") putc("I") putc("L") }
298         }
299
300         ; --- step 0, partition entry 1 (offset 446 = buf_hi[190]) ------------
301         if (fail=0) {
302            if (step=0) {
303               ptype:=buf_hi[194]               ;offset 450
304               c3:=buf_hi[201]                  ;start LBA, offset 454..457, little-endian
305               c2:=buf_hi[200]
306               c1:=buf_hi[199]
307               c0:=buf_hi[198]
308               p_lba:=mk32(c3,c2,c1,c0)
309               putc("Space") putc("T") putc("Y") putc("P") putc("E") putc("Equal")
310               put_hex(ptype)
311               putc("Space") putc("L") putc("B") putc("A") putc("Equal")
312               put_hex32(p_lba)
313               put_crlf()
314               if (ptype=11) { }                ;0B = FAT32 (CHS)
315               altif (ptype=12) { }             ;0C = FAT32 (LBA)
316               default { fail:=1 }
317               lba:=p_lba                       ;step 1 reads the boot sector
318            }
319
320            ; --- step 1, BIOS parameter block (all offsets < 256 = buf_lo) ----
321            default {
322               t:=buf_lo[82]                    ;"FAT32" at offset 82..86
```

```
            s1:=buf_lo[86]
            if (t=70) {                     ;F
               if (s1=50) {                 ;2
                  putc("Space") putc("F") putc("A") putc("T") putc("_threE") putc("_twO")
               }
               default { fail:=1 }
            }
            default { fail:=1 }
            put_crlf()

            ; BPS = BytsPerSec, must be 0200 (512)
            putc("B") putc("P") putc("S") putc("Equal")
            t:=buf_lo[12]
            put_hex(t)
            t:=buf_lo[11]
            put_hex(t)
            s1:=buf_lo[12]
            s2:=buf_lo[11]
            if (s1!=2) { fail:=1 }
            if (s2!=0) { fail:=1 }
            ; SPC = SecPerClus
            putc("Space") putc("S") putc("P") putc("C") putc("Equal")
            t:=buf_lo[13]
            put_hex(t)
            ; RSV = RsvdSecCnt
            putc("Space") putc("R") putc("S") putc("V") putc("Equal")
            t:=buf_lo[15]
            put_hex(t)
            t:=buf_lo[14]
            put_hex(t)
            ; NF = NumFATs
            nf:=buf_lo[16]
            putc("Space") putc("N") putc("F") putc("Equal")
            put_hex(nf)
            put_crlf()

            ; FSZ = FATSz32 (sectors per FAT), ROOT = RootClus
            c3:=buf_lo[39]
            c2:=buf_lo[38]
            c1:=buf_lo[37]
            c0:=buf_lo[36]
            fsz:=mk32(c3,c2,c1,c0)
            putc("F") putc("S") putc("Z") putc("Equal")
            put_hex32(fsz)
            putc("Space") putc("R") putc("O") putc("O") putc("T") putc("Equal")
            t:=buf_lo[47]
            put_hex(t)
            t:=buf_lo[46]
            put_hex(t)
            t:=buf_lo[45]
            put_hex(t)
            t:=buf_lo[44]
            put_hex(t)
            put_crlf()

            ; fat_lba = p_lba + RsvdSecCnt ; data_lba = fat_lba + NumFATs * FATSz32
            c1:=buf_lo[15]
            c0:=buf_lo[14]
            rsvd:=mk32(0,0,c1,c0)
            fat_lba:=p_lba+rsvd
            data_lba:=fat_lba
            i:=0
            loop {                          ;multiply by repeated addition (nf is 1 or 2)
               if (i>=nf) { quit }
               data_lba:=data_lba+fsz
               i:=i+1
            }
            putc("F") putc("A") putc("T") putc("Equal")
            put_hex32(fat_lba)
            putc("Space") putc("D") putc("A") putc("T") putc("A") putc("Equal")
            put_hex32(data_lba)
            put_crlf()
         }
      }
      default { put_crlf() }
   }

   if (fail=0) { setbit(LEDPORT,GREEN) }
   default { setbit(LEDPORT,RED) }
   put_crlf()
   delay(5000)
}
```

A.5 Test program `18_sd_5_dir.jan`

```
; Program name: 18_sd_5_dir.jan
; Object file name: 18_sd_5_dir.asm
; Uses: ascii.jan
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 5 (root directory)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card (FAT32, MBR) inserted.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 16/10/2026
; Version: 0.1
; Description: lists the root directory (8.3 entries) via UART.
;   The program is a STATE MACHINE with ONE sector read per pass of the main loop
;   state 0 = MBR          -> p_lba, next state 1
;   state 1 = boot sector  -> spc, fat_lba, data_lba, c = RootClus, s = 0, next state 2
;   state 2 = directory sector s of cluster c, LBA = data_lba + (c-2)*spc + s
;             16 entries of 32 bytes; first byte 00 = end of directory (state 9),
;             E5 = deleted, attribute 0F = long-file-name part (both skipped)
;             after the last sector of the cluster -> state 3
;   state 3 = FAT sector holding the entry of cluster c, LBA = fat_lba + c/128,
;             entry = 4 bytes at offset (c mod 128)*4, upper 4 bits masked
;             >= 0FFFFFF8 = end of the chain (state 9), else c = next, s = 0, state 2
;   state 9 = done
; Output per entry
;   NAME    .EXT A=20 C=00000003 S=0000001A
;   (A = attribute hex, 10 = directory, 08 = volume label, 20 = archive;
;    C = first cluster; S = size in bytes, hex)
; Green LED = listing complete, red LED = error. Repeats every 10 s.
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;card deselected before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ---------------------------------------------------------------------------
; helpers (from blocks B2..B4)
; ---------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
   return(r)
}

; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                     ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                           ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }           ;no answer -> r stays FF
   }
   return(r)
}

; CS high, then 8 more clocks so the card releases MISO
subroutine sd_deselect() {
   raw8 r
```

```
   setbit(CSPORT,CS)
   r:=xchg(255)
}

subroutine putc(raw8 ch) {
   .uart_tx_byte(ch,context,local_id)
}

subroutine put_hex(raw8 v) {
   raw8 d,c
   d:=v/16
   c:=digit_to_ascii(d)
   putc(c)
   d:=v%16
   c:=digit_to_ascii(d)
   putc(c)
}

subroutine put_crlf() {
   putc("CR")
   putc("LF")
}

; (since B4) four bytes (most significant first) -> one raw32 value
function raw32 mk32(raw8 b3,raw8 b2,raw8 b1,raw8 b0) {
   raw32 v
   v:=raw32(b3)
   v:=v*256
   v:=v+raw32(b2)
   v:=v*256
   v:=v+raw32(b1)
   v:=v*256
   v:=v+raw32(b0)
   return(v)
}

; (since B4) raw32 -> 8 hex digits; raw8(w) keeps the low byte (narrowing), w/256 drops it
subroutine put_hex32(raw32 v) {
   raw32 w
   raw8 b3,b2,b1,b0
   w:=v
   b0:=raw8(w)
   w:=w/256
   b1:=raw8(w)
   w:=w/256
   b2:=raw8(w)
   w:=w/256
   b3:=raw8(w)
   put_hex(b3)
   put_hex(b2)
   put_hex(b1)
   put_hex(b0)
}

; ------------------------------------------------------------------------
; block B2 as one silent function (unchanged since B3).
; returns 0 = initialization failed, 1 = SDSC (byte addresses), 2 = SDHC (block addresses)
; ------------------------------------------------------------------------
function raw8 sd_init() {
   raw8 r,k,st,tries,v2,hcs,crc41,last,ocr0,kind
   st:=0
   kind:=0
   setbit(CSPORT,CS)
   .spi_set_speed(SPI_SLOW)
   take k from [0,10) { r:=xchg(255) }       ;80 clocks with CS high

   clearbit(CSPORT,CS)
   r:=sd_cmd(64,0,0,0,0,149)                 ;CMD0 -> expect 01
   sd_deselect()
   if (r!=1) { st:=1 }

   v2:=0
   if (st=0) {
      clearbit(CSPORT,CS)
      r:=sd_cmd(72,0,0,1,170,135)            ;CMD8 -> expect 01 + 000001AA
      take k from [0,4) { last:=xchg(255) } ;keep only the echo byte AA
      sd_deselect()
      if (r=1) {
         if (last=170) { v2:=1 }
         default { st:=1 }
      }
      altif (r=5) { }                        ;V1 card, no R7
      default { st:=1 }
   }

   if (st=0) {
      if (v2=1) { hcs:=64  crc41:=119 }
```

```
        default { hcs:=0  crc41:=229 }
        tries:=0
        loop {
           clearbit(CSPORT,CS)
           r:=sd_cmd(119,0,0,0,0,101)          ;CMD55
           r:=sd_cmd(105,hcs,0,0,0,crc41)      ;ACMD41
           sd_deselect()
           if (r=0) { quit }
           tries:=tries+1
           if (tries>=200) { quit }
           delay(10)
        }
        if (r!=0) { st:=1 }
     }

     if (st=0) {
        clearbit(CSPORT,CS)
        r:=sd_cmd(122,0,0,0,0,253)             ;CMD58 -> OCR
        ocr0:=xchg(255)                        ;first OCR byte holds CCS (bit 6)
        take k from [0,3) { last:=xchg(255) }
        sd_deselect()
        if (r!=0) { st:=1 }
     }

     if (st=0) {
        ocr0:=ocr0&64
        if (ocr0!=0) { kind:=2 }
        default { kind:=1 }
        .spi_set_speed(SPI_FAST)               ;4 MHz for the data transfer
     }
     return(kind)
  }

  ; ------------------------------------------------------------------------
  ; main
  ; ------------------------------------------------------------------------
  raw8 r,k,t,mrk,fail,kind,state,s1,s2,ptype,nf,i,a3,a2,a1,a0,c3,c2,c1,c0
  raw8 spc,s,e,j,half,base,idx,q,attr,nent,ncl
  raw16 n
  raw32 lba,adr,p_lba,rsvd,fsz,fat_lba,data_lba,c,pm,size
  array raw8 buf_lo[256]              ;sector bytes   0..255
  array raw8 buf_hi[256]              ;sector bytes 256..511
  array raw8 ent[32]                  ;copy of the current 32-byte directory entry

  loop {
     fail:=0
     clearbit(LEDPORT,GREEN)
     clearbit(LEDPORT,RED)

     kind:=sd_init()
     if (kind=0) {
        fail:=1
        putc("I") putc("N") putc("I") putc("T") putc("Space") putc("F") putc("A") putc("I")
       putc("L")
        put_crlf()
     }

     state:=0
     lba:=0
     nent:=0
     ncl:=0
     loop {
        if (fail!=0) { quit }

        ; --- state 2 computes its sector number here --------------------------
        if (state=2) {
           pm:=c-2
           lba:=0
           i:=0
           loop {                              ;lba = (c-2)*spc by repeated addition
              if (i>=spc) { quit }
              lba:=lba+pm
              i:=i+1
           }
           lba:=lba+data_lba
           lba:=lba+raw32(s)
        }

        ; --- the ONE sector read (as in block B4) -----------------------------
        adr:=lba
        if (kind=1) { adr:=lba*512 }           ;SDSC = byte address
        a0:=raw8(adr)
        adr:=adr/256
        a1:=raw8(adr)
        adr:=adr/256
        a2:=raw8(adr)
        adr:=adr/256
```

```
        a3:=raw8(adr)
        mrk:=255
        clearbit(CSPORT,CS)
        r:=sd_cmd(81,a3,a2,a1,a0,255)
        if (r=0) {
           n:=0
           loop {
              mrk:=xchg(255)
              if (mrk!=255) { quit }
              n:=n+1
              if (n>=50000) { quit }
           }
        }
        if (mrk=254) {
           k:=0
           loop {
              t:=xchg(255)
              buf_lo[k]:=t
              k:=k+1
              if (k=0) { quit }
           }
           loop {
              t:=xchg(255)
              buf_hi[k]:=t
              k:=k+1
              if (k=0) { quit }
           }
           t:=xchg(255)                          ;CRC16, ignored
           t:=xchg(255)
        }
        sd_deselect()
        if (mrk!=254) {
           fail:=1
           putc("R") putc("E") putc("A") putc("D") putc("Space") putc("F") putc("A") putc("I")
       putc("L")
           put_crlf()
           quit
        }

        ; --- evaluate the sector according to the state -----------------------
        if (state=0) {                           ;MBR, partition entry 1
           ptype:=buf_hi[194]
           c3:=buf_hi[201]
           c2:=buf_hi[200]
           c1:=buf_hi[199]
           c0:=buf_hi[198]
           p_lba:=mk32(c3,c2,c1,c0)
           if (ptype=11) { }
           altif (ptype=12) { }
           default { fail:=1 }
           lba:=p_lba
           state:=1
        }
        altif (state=1) {                        ;boot sector, BPB
           spc:=buf_lo[13]
           nf:=buf_lo[16]
           c1:=buf_lo[15]
           c0:=buf_lo[14]
           rsvd:=mk32(0,0,c1,c0)
           c3:=buf_lo[39]
           c2:=buf_lo[38]
           c1:=buf_lo[37]
           c0:=buf_lo[36]
           fsz:=mk32(c3,c2,c1,c0)
           fat_lba:=p_lba+rsvd
           data_lba:=fat_lba
           i:=0
           loop {
              if (i>=nf) { quit }
              data_lba:=data_lba+fsz
              i:=i+1
           }
           c3:=buf_lo[47]                        ;RootClus
           c2:=buf_lo[46]
           c1:=buf_lo[45]
           c0:=buf_lo[44]
           c:=mk32(c3,c2,c1,c0)
           s:=0
           if (spc=0) { fail:=1 }
           putc("R") putc("O") putc("O") putc("T") putc("Space") putc("C") putc("Equal")
           put_hex32(c)
           put_crlf()
           state:=2
        }
        altif (state=2) {                        ;one directory sector = 16 entries
           e:=0
           loop {
```

```
            half:=e/8                       ;entries 0..7 in buf_lo, 8..15 in buf_hi
            base:=e%8
            base:=base*32
            take j from [0,32) {            ;copy the entry to ent[0..31]
               idx:=base+j
               if (half=0) { t:=buf_lo[idx] }
               default { t:=buf_hi[idx] }
               ent[j]:=t
            }
            t:=ent[0]
            attr:=ent[11]
            if (t=0) {                      ;00 = no more entries
               state:=9
               quit
            }
            if (t=229) { }                  ;E5 = deleted entry
            altif (attr=15) { }             ;0F = part of a long file name
            default {
               take j from [0,8) {          ;name, padded with spaces
                  t:=ent[j]
                  putc(t)
               }
               putc("Dot")
               take j from [8,11) {         ;extension
                  t:=ent[j]
                  putc(t)
               }
               putc("Space") putc("A") putc("Equal")
               put_hex(attr)
               c3:=ent[21]                  ;first cluster, high word at 20/21
               c2:=ent[20]
               c1:=ent[27]                  ;low word at 26/27
               c0:=ent[26]
               pm:=mk32(c3,c2,c1,c0)
               putc("Space") putc("C") putc("Equal")
               put_hex32(pm)
               c3:=ent[31]                  ;size, offset 28..31
               c2:=ent[30]
               c1:=ent[29]
               c0:=ent[28]
               size:=mk32(c3,c2,c1,c0)
               putc("Space") putc("S") putc("Equal")
               put_hex32(size)
               t:=attr&16
               if (t!=0) { putc("Space") putc("Less") putc("D") putc("I") putc("R") putc("
Greater") }
               t:=attr&8
               if (t!=0) { putc("Space") putc("Less") putc("V") putc("O") putc("L") putc("
Greater") }
               put_crlf()
               nent:=nent+1
            }
            e:=e+1
            if (e>=16) { quit }
         }
         if (state=2) {                     ;not finished -> next sector or FAT
            s:=s+1
            if (s>=spc) {
               pm:=c/128                    ;128 FAT entries of 4 bytes per sector
               lba:=fat_lba+pm
               state:=3
            }
         }
      }
      default {                             ;state 3 = FAT sector, find the next cluster
         q:=raw8(c)
         q:=q&127                           ;c mod 128 = entry number in this sector
         half:=q/64                         ;entries 0..63 in buf_lo, 64..127 in buf_hi
         idx:=q&63
         idx:=idx*4                         ;byte offset of the 4-byte entry
         if (half=0) {
            c0:=buf_lo[idx]
            idx:=idx+1
            c1:=buf_lo[idx]
            idx:=idx+1
            c2:=buf_lo[idx]
            idx:=idx+1
            c3:=buf_lo[idx]
         }
         default {
            c0:=buf_hi[idx]
            idx:=idx+1
            c1:=buf_hi[idx]
            idx:=idx+1
            c2:=buf_hi[idx]
            idx:=idx+1
            c3:=buf_hi[idx]
```

```
            }
            c3:=c3&15                           ;FAT32 entries use only 28 bits
            c:=mk32(c3,c2,c1,c0)
            ncl:=ncl+1
            if (c>=268435448) { state:=9 }      ;0FFFFFF8.. = end of the cluster chain
            altif (c<2) { fail:=1 }             ;free or reserved = broken chain
            altif (ncl>=200) { fail:=1 }        ;safety limit against a looping chain
            default {
               s:=0
               state:=2
            }
         }

         if (state=9) { quit }
      }

      ; --- summary ---------------------------------------------------------
      putc("N") putc("Equal")
      put_hex(nent)
      putc("Space")
      if (fail=0) {
         putc("O") putc("K")
         setbit(LEDPORT,GREEN)
      }
      default {
         putc("F") putc("A") putc("I") putc("L")
         setbit(LEDPORT,RED)
      }
      put_crlf()
      put_crlf()
      delay(10000)
   }
```

A.6 Test program `18_sd_6_read_file.jan`

```
; Program name: 18_sd_6_read_file.jan
; Object file name: 18_sd_6_read_file.asm
; Uses: ascii.jan
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 6 (read a file)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
;                RC6 (TX), RC7 (RX); USART asynchronous 38400 Bd (NOT 9600!)
; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card (FAT32, MBR) inserted.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 16/10/2026
; Version: 0.1
; Description: searches the root directory for the 8.3 name in array target
;   (here DATA_V~2.TXT, stored as 11 bytes "DATA_V~2TXT" without the dot), follows
;   the file's cluster chain through the FAT and sends its content via UART.
;   State machine of block B5, extended
;   state 0 = MBR, state 1 = boot sector (as in B5)
;   state 2 = directory sector, compare every entry with target
;             found -> c = first cluster, rem = size, fileon = 1, s = 0, state 4
;             entry 00 -> NOT FOUND
;   state 3 = FAT lookup of cluster c (as in B5); continues with state 2 (directory)
;             or state 4 (file) according to fileon
;   state 4 = file sector s of cluster c, send min(512, rem) bytes, rem decreases;
;             rem = 0 -> done (state 9)
;   The UART runs at 38400 Bd (SPBRG 25, error 0.16 %): 127134 bytes take ~33 s
;   (at 9600 Bd it would be over 2 minutes).
; Output
;   FOUND C=0000000B S=0001F09E
;   ----
;   <the file content, as stored>
;   ----
;   END N=0001F09E SUM=xxxx CL=20
;   (N = bytes sent, SUM = 16-bit sum of all bytes, CL = clusters read, hex)
; Check on the PC, e.g. with Python
;   d = open("DATA_V~2.TXT","rb").read(); print(hex(len(d)), hex(sum(d) & 0xFFFF))
; Runs once; green LED = file sent completely, red LED = error.
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>    ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>           ;RA1
<define> GREEN:=0 </define>         ;RA0
<define> CSPORT:=3979 </define>     ;LATC 0xF8B
```

```
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                   ;card deselected before SPI starts
.configure_uart(38400,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ------------------------------------------------------------------------
; helpers (from blocks B2..B4)
; ------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
   return(r)
}

; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                      ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                            ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }            ;no answer -> r stays FF
   }
   return(r)
}

; CS high, then 8 more clocks so the card releases MISO
subroutine sd_deselect() {
   raw8 r
   setbit(CSPORT,CS)
   r:=xchg(255)
}

subroutine putc(raw8 ch) {
   .uart_tx_byte(ch,context,local_id)
}

subroutine put_hex(raw8 v) {
   raw8 d,c
   d:=v/16
   c:=digit_to_ascii(d)
   putc(c)
   d:=v%16
   c:=digit_to_ascii(d)
   putc(c)
}

subroutine put_crlf() {
   putc("CR")
   putc("LF")
}

; (since B4) four bytes (most significant first) -> one raw32 value
function raw32 mk32(raw8 b3,raw8 b2,raw8 b1,raw8 b0) {
   raw32 v
   v:=raw32(b3)
   v:=v*256
   v:=v+raw32(b2)
   v:=v*256
   v:=v+raw32(b1)
   v:=v*256
   v:=v+raw32(b0)
   return(v)
}

; (since B4) raw32 -> 8 hex digits; raw8(w) keeps the low byte (narrowing), w/256 drops it
subroutine put_hex32(raw32 v) {
   raw32 w
   raw8 b3,b2,b1,b0
   w:=v
   b0:=raw8(w)
   w:=w/256
   b1:=raw8(w)
   w:=w/256
```

```
   b2:=raw8(w)
   w:=w/256
   b3:=raw8(w)
   put_hex(b3)
   put_hex(b2)
   put_hex(b1)
   put_hex(b0)
}

; ------------------------------------------------------------------------
; block B2 as one silent function (unchanged since B3).
; returns 0 = initialization failed, 1 = SDSC (byte addresses), 2 = SDHC (block addresses)
; ------------------------------------------------------------------------
function raw8 sd_init() {
   raw8 r,k,st,tries,v2,hcs,crc41,last,ocr0,kind
   st:=0
   kind:=0
   setbit(CSPORT,CS)
   .spi_set_speed(SPI_SLOW)
   take k from [0,10) { r:=xchg(255) }       ;80 clocks with CS high

   clearbit(CSPORT,CS)
   r:=sd_cmd(64,0,0,0,0,149)                 ;CMD0 -> expect 01
   sd_deselect()
   if (r!=1) { st:=1 }

   v2:=0
   if (st=0) {
      clearbit(CSPORT,CS)
      r:=sd_cmd(72,0,0,1,170,135)            ;CMD8 -> expect 01 + 000001AA
      take k from [0,4) { last:=xchg(255) } ;keep only the echo byte AA
      sd_deselect()
      if (r=1) {
         if (last=170) { v2:=1 }
         default { st:=1 }
      }
      altif (r=5) { }                        ;V1 card, no R7
      default { st:=1 }
   }

   if (st=0) {
      if (v2=1) { hcs:=64  crc41:=119 }
      default { hcs:=0  crc41:=229 }
      tries:=0
      loop {
         clearbit(CSPORT,CS)
         r:=sd_cmd(119,0,0,0,0,101)          ;CMD55
         r:=sd_cmd(105,hcs,0,0,0,crc41)      ;ACMD41
         sd_deselect()
         if (r=0) { quit }
         tries:=tries+1
         if (tries>=200) { quit }
         delay(10)
      }
      if (r!=0) { st:=1 }
   }

   if (st=0) {
      clearbit(CSPORT,CS)
      r:=sd_cmd(122,0,0,0,0,253)             ;CMD58 -> OCR
      ocr0:=xchg(255)                        ;first OCR byte holds CCS (bit 6)
      take k from [0,3) { last:=xchg(255) }
      sd_deselect()
      if (r!=0) { st:=1 }
   }

   if (st=0) {
      ocr0:=ocr0&64
      if (ocr0!=0) { kind:=2 }
      default { kind:=1 }
      .spi_set_speed(SPI_FAST)               ;4 MHz for the data transfer
   }
   return(kind)
}

; ------------------------------------------------------------------------
; main
; ------------------------------------------------------------------------
raw8 r,k,t,u,mrk,fail,kind,state,ptype,nf,i,a3,a2,a1,a0,c3,c2,c1,c0
raw8 spc,s,e,j,half,base,idx,q,attr,fileon,match
raw16 n,ncl,sum
raw32 lba,adr,p_lba,rsvd,fsz,fat_lba,data_lba,c,pm,fsize,rem,nsent
array raw8 buf_lo[256]             ;sector bytes   0..255
array raw8 buf_hi[256]             ;sector bytes 256..511
array raw8 ent[32]                 ;copy of the current 32-byte directory entry
array raw8 target[11]:=["D","A","T","A","Underscore","V","Tilde","_tw0","T","X","T"]

```

```
fail:=0
clearbit(LEDPORT,GREEN)
clearbit(LEDPORT,RED)

kind:=sd_init()
if (kind=0) {
   fail:=1
   putc("I") putc("N") putc("I") putc("T") putc("Space") putc("F") putc("A") putc("I") putc("
     L")
   put_crlf()
}

state:=0
lba:=0
fileon:=0
ncl:=0
sum:=0
nsent:=0
rem:=0
loop {
   if (fail!=0) { quit }

   ; --- states 2 and 4 compute their sector number here ----------------------
   if (state>=2) {
      if (state!=3) {
         pm:=c-2
         lba:=0
         i:=0
         loop {                                ;lba = (c-2)*spc by repeated addition
            if (i>=spc) { quit }
            lba:=lba+pm
            i:=i+1
         }
         lba:=lba+data_lba
         lba:=lba+raw32(s)
      }
   }

   ; --- the ONE sector read (as in blocks B4/B5) -----------------------------
   adr:=lba
   if (kind=1) { adr:=lba*512 }             ;SDSC = byte address
   a0:=raw8(adr)
   adr:=adr/256
   a1:=raw8(adr)
   adr:=adr/256
   a2:=raw8(adr)
   adr:=adr/256
   a3:=raw8(adr)
   mrk:=255
   clearbit(CSPORT,CS)
   r:=sd_cmd(81,a3,a2,a1,a0,255)
   if (r=0) {
      n:=0
      loop {
         mrk:=xchg(255)
         if (mrk!=255) { quit }
         n:=n+1
         if (n>=50000) { quit }
      }
   }
   if (mrk=254) {
      k:=0
      loop {
         t:=xchg(255)
         buf_lo[k]:=t
         k:=k+1
         if (k=0) { quit }
      }
      loop {
         t:=xchg(255)
         buf_hi[k]:=t
         k:=k+1
         if (k=0) { quit }
      }
      t:=xchg(255)                           ;CRC16, ignored
      t:=xchg(255)
   }
   sd_deselect()
   if (mrk!=254) {
      fail:=1
      putc("R") putc("E") putc("A") putc("D") putc("Space") putc("F") putc("A") putc("I")
     putc("L")
      put_crlf()
      quit
   }

   ; --- evaluate the sector according to the state ---------------------------
```

```
    if (state=0) {                          ;MBR, partition entry 1
       ptype:=buf_hi[194]
       c3:=buf_hi[201]
       c2:=buf_hi[200]
       c1:=buf_hi[199]
       c0:=buf_hi[198]
       p_lba:=mk32(c3,c2,c1,c0)
       if (ptype=11) { }
       altif (ptype=12) { }
       default { fail:=1 }
       lba:=p_lba
       state:=1
    }
    altif (state=1) {                       ;boot sector, BPB
       spc:=buf_lo[13]
       nf:=buf_lo[16]
       c1:=buf_lo[15]
       c0:=buf_lo[14]
       rsvd:=mk32(0,0,c1,c0)
       c3:=buf_lo[39]
       c2:=buf_lo[38]
       c1:=buf_lo[37]
       c0:=buf_lo[36]
       fsz:=mk32(c3,c2,c1,c0)
       fat_lba:=p_lba+rsvd
       data_lba:=fat_lba
       i:=0
       loop {
          if (i>=nf) { quit }
          data_lba:=data_lba+fsz
          i:=i+1
       }
       c3:=buf_lo[47]                       ;RootClus
       c2:=buf_lo[46]
       c1:=buf_lo[45]
       c0:=buf_lo[44]
       c:=mk32(c3,c2,c1,c0)
       s:=0
       if (spc=0) { fail:=1 }
       state:=2
    }
    altif (state=2) {                       ;directory sector, search target
       e:=0
       loop {
          half:=e/8
          base:=e%8
          base:=base*32
          take j from [0,32) {
             idx:=base+j
             if (half=0) { t:=buf_lo[idx] }
             default { t:=buf_hi[idx] }
             ent[j]:=t
          }
          t:=ent[0]
          if (t=0) {                        ;00 = end of directory, name not there
             putc("N") putc("O") putc("T") putc("Space") putc("F") putc("O") putc("U") putc("N
  ") putc("D")
             put_crlf()
             fail:=1
             quit
          }
          match:=1
          take j from [0,11) {              ;compare the 11 name bytes
             t:=ent[j]
             u:=target[j]
             if (t!=u) { match:=0 }
          }
          attr:=ent[11]
          if (attr=15) { match:=0 }         ;never a long-name piece
          t:=attr&16
          if (t!=0) { match:=0 }            ;never a directory
          if (match=1) {
             c3:=ent[21]                    ;first cluster
             c2:=ent[20]
             c1:=ent[27]
             c0:=ent[26]
             c:=mk32(c3,c2,c1,c0)
             c3:=ent[31]                    ;size
             c2:=ent[30]
             c1:=ent[29]
             c0:=ent[28]
             fsize:=mk32(c3,c2,c1,c0)
             rem:=fsize
             putc("F") putc("O") putc("U") putc("N") putc("D")
             putc("Space") putc("C") putc("Equal")
             put_hex32(c)
             putc("Space") putc("S") putc("Equal")
```

```
            put_hex32(fsize)
            put_crlf()
            putc("Minus") putc("Minus") putc("Minus") putc("Minus")
            put_crlf()
            fileon:=1
            ncl:=0                          ;from now on count the file's clusters
            s:=0
            if (rem=0) { state:=9 }         ;empty file, no cluster to read
            default { state:=4 }
            quit
         }
         e:=e+1
         if (e>=16) { quit }
      }
      if (state=2) {                        ;target not in this sector
         s:=s+1
         if (s>=spc) {
            pm:=c/128
            lba:=fat_lba+pm
            state:=3
         }
      }
   }
   altif (state=4) {                        ;file sector, send min(512, rem) bytes
      k:=0
      loop {                                ;bytes 0..255
         if (rem=0) { quit }
         t:=buf_lo[k]
         putc(t)
         sum:=sum+raw16(t)
         rem:=rem-1
         nsent:=nsent+1
         k:=k+1
         if (k=0) { quit }
      }
      k:=0
      loop {                                ;bytes 256..511
         if (rem=0) { quit }
         t:=buf_hi[k]
         putc(t)
         sum:=sum+raw16(t)
         rem:=rem-1
         nsent:=nsent+1
         k:=k+1
         if (k=0) { quit }
      }
      if (rem=0) { state:=9 }
      default {
         s:=s+1
         if (s>=spc) {
            pm:=c/128
            lba:=fat_lba+pm
            state:=3
         }
      }
   }
   default {                                ;state 3 = FAT sector, next cluster
      q:=raw8(c)
      q:=q&127
      half:=q/64
      idx:=q&63
      idx:=idx*4
      if (half=0) {
         c0:=buf_lo[idx]
         idx:=idx+1
         c1:=buf_lo[idx]
         idx:=idx+1
         c2:=buf_lo[idx]
         idx:=idx+1
         c3:=buf_lo[idx]
      }
      default {
         c0:=buf_hi[idx]
         idx:=idx+1
         c1:=buf_hi[idx]
         idx:=idx+1
         c2:=buf_hi[idx]
         idx:=idx+1
         c3:=buf_hi[idx]
      }
      c3:=c3&15
      c:=mk32(c3,c2,c1,c0)
      ncl:=ncl+1
      if (c>=268435448) {                   ;end of the chain before the end of the data
         fail:=1
         putc("C") putc("H") putc("A") putc("I") putc("N") putc("Space") putc("E") putc("N")
       putc("D")
```

```
481             put_crlf()
482          }
483          altif (c<2) { fail:=1 }                ;free or reserved = broken chain
484          altif (ncl>=60000) { fail:=1 }         ;safety limit
485          default {
486             s:=0
487             if (fileon=1) { state:=4 }
488             default { state:=2 }
489          }
490       }
491
492       if (state=9) { quit }
493    }
494
495    ; --- summary ----------------------------------------------------------
496    if (fileon=1) {
497       put_crlf()
498       putc("Minus") putc("Minus") putc("Minus") putc("Minus")
499       put_crlf()
500       putc("E") putc("N") putc("D") putc("Space") putc("N") putc("Equal")
501       put_hex32(nsent)
502       putc("Space") putc("S") putc("U") putc("M") putc("Equal")
503       t:=high(sum)
504       put_hex(t)
505       t:=low(sum)
506       put_hex(t)
507       putc("Space") putc("C") putc("L") putc("Equal")
508       t:=low(ncl)
509       t:=t+1                                 ;clusters read = FAT lookups + the first one
510       put_hex(t)
511       put_crlf()
512    }
513    if (fail=0) { setbit(LEDPORT,GREEN) }
514    default { setbit(LEDPORT,RED) }
515    loop { delay(1000) }                      ;done, wait for reset
```

A.7 Test program `18_sd_7_write.jan`

```
1  ; Program name: 18_sd_7_write.jan
2  ; Object file name: 18_sd_7_write.asm
3  ; Uses: ascii.jan
4  ; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
5  ; Project: FAT32 + SD card on the PIC18F452, Baustein 7 (write one sector)
6  ; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
7  ;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
8  ;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
9  ; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card (MBR) inserted.
10 ; Author: Claude Baumann
11 ; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
12 ; Date: 16/10/2026
13 ; Version: 0.1
14 ; Description: first WRITE access. To leave the file system untouched, the test
15 ;   sector is the last sector of the unused gap between the MBR (LBA 0) and the
16 ;   partition start, tlba = p_lba - 1 (127 on the author's card). The test only
17 ;   writes if that sector is completely zero, and writes zeros back at the end.
18 ;   Six operations, one read site and one write site (take op from [0,6))
19 ;   op 0  read  LBA 0       MBR -> p_lba, tlba = p_lba - 1 (p_lba >= 2 required)
20 ;   op 1  read  tlba        must be all zero, else STOP (nothing is written)
21 ;   op 2  write tlba        pattern buf_lo[k] = k, buf_hi[k] = 255 - k
22 ;   op 3  read  tlba        compare with the pattern -> ERR = number of wrong bytes
23 ;   op 4  write tlba        zeros (restore)
24 ;   op 5  read  tlba        must be all zero again
25 ;   CMD24 (WRITE_BLOCK) sequence = command -> R1 = 00 -> one FF byte -> start token FE
26 ;   -> 512 data bytes -> 2 CRC bytes (not checked) -> data response xxx0sss1, where
27 ;   sss = 010 (low bits 05) means "accepted" -> card holds MISO low (00) while busy.
28 ; Expected output
29 ;   INIT OK SDSC
30 ;   TEST LBA=0000007F
31 ;   ZERO OK
32 ;   WRITE R=05 BUSY=xxxx
33 ;   VERIFY ERR=0000
34 ;   WRITE R=05 BUSY=xxxx
35 ;   RESTORE OK
36 ;   (BUSY = number of FF polls until the card had finished writing, hex)
37 ; Runs once; green LED = all steps passed, red LED = error.
38 ;-------------------------------------------------------------
39
40 <insert> _specific_pic18F452.jan </insert>
41 <insert> _pic18Fxx_.jan </insert>
42 <insert> iCode_Microchip_18F452.jan </insert>
43 <insert> ascii.jan </insert>
44 <define> FREQUENCY:=16000000 </define>
45
46 <define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
```

```
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;card deselected before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ------------------------------------------------------------------------
; helpers (from blocks B2..B4)
; ------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
   return(r)
}

; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                     ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                           ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }           ;no answer -> r stays FF
   }
   return(r)
}

; CS high, then 8 more clocks so the card releases MISO
subroutine sd_deselect() {
   raw8 r
   setbit(CSPORT,CS)
   r:=xchg(255)
}

subroutine putc(raw8 ch) {
   .uart_tx_byte(ch,context,local_id)
}

subroutine put_hex(raw8 v) {
   raw8 d,c
   d:=v/16
   c:=digit_to_ascii(d)
   putc(c)
   d:=v%16
   c:=digit_to_ascii(d)
   putc(c)
}

subroutine put_crlf() {
   putc("CR")
   putc("LF")
}

; (since B4) four bytes (most significant first) -> one raw32 value
function raw32 mk32(raw8 b3,raw8 b2,raw8 b1,raw8 b0) {
   raw32 v
   v:=raw32(b3)
   v:=v*256
   v:=v+raw32(b2)
   v:=v*256
   v:=v+raw32(b1)
   v:=v*256
   v:=v+raw32(b0)
   return(v)
}

; (since B4) raw32 -> 8 hex digits; raw8(w) keeps the low byte (narrowing), w/256 drops it
subroutine put_hex32(raw32 v) {
   raw32 w
   raw8 b3,b2,b1,b0
   w:=v
   b0:=raw8(w)
```

```
    w:=w/256
    b1:=raw8(w)
    w:=w/256
    b2:=raw8(w)
    w:=w/256
    b3:=raw8(w)
    put_hex(b3)
    put_hex(b2)
    put_hex(b1)
    put_hex(b0)
}

; ------------------------------------------------------------------------
; block B2 as one silent function (unchanged since B3).
; returns 0 = initialization failed, 1 = SDSC (byte addresses), 2 = SDHC (block addresses)
; ------------------------------------------------------------------------
function raw8 sd_init() {
   raw8 r,k,st,tries,v2,hcs,crc41,last,ocr0,kind
   st:=0
   kind:=0
   setbit(CSPORT,CS)
   .spi_set_speed(SPI_SLOW)
   take k from [0,10) { r:=xchg(255) }       ;80 clocks with CS high

   clearbit(CSPORT,CS)
   r:=sd_cmd(64,0,0,0,0,149)                 ;CMD0 -> expect 01
   sd_deselect()
   if (r!=1) { st:=1 }

   v2:=0
   if (st=0) {
      clearbit(CSPORT,CS)
      r:=sd_cmd(72,0,0,1,170,135)            ;CMD8 -> expect 01 + 000001AA
      take k from [0,4) { last:=xchg(255) } ;keep only the echo byte AA
      sd_deselect()
      if (r=1) {
         if (last=170) { v2:=1 }
         default { st:=1 }
      }
      altif (r=5) { }                        ;V1 card, no R7
      default { st:=1 }
   }

   if (st=0) {
      if (v2=1) { hcs:=64  crc41:=119 }
      default { hcs:=0  crc41:=229 }
      tries:=0
      loop {
         clearbit(CSPORT,CS)
         r:=sd_cmd(119,0,0,0,0,101)          ;CMD55
         r:=sd_cmd(105,hcs,0,0,0,crc41)      ;ACMD41
         sd_deselect()
         if (r=0) { quit }
         tries:=tries+1
         if (tries>=200) { quit }
         delay(10)
      }
      if (r!=0) { st:=1 }
   }

   if (st=0) {
      clearbit(CSPORT,CS)
      r:=sd_cmd(122,0,0,0,0,253)             ;CMD58 -> OCR
      ocr0:=xchg(255)                        ;first OCR byte holds CCS (bit 6)
      take k from [0,3) { last:=xchg(255) }
      sd_deselect()
      if (r!=0) { st:=1 }
   }

   if (st=0) {
      ocr0:=ocr0&64
      if (ocr0!=0) { kind:=2 }
      default { kind:=1 }
      .spi_set_speed(SPI_FAST)               ;4 MHz for the data transfer
   }
   return(kind)
}

; ------------------------------------------------------------------------
; main
; ------------------------------------------------------------------------
raw8 r,k,t,u,mrk,fail,verr,kind,op,wr,wres,ptype,a3,a2,a1,a0,c3,c2,c1,c0
raw16 n,busy,errs
raw32 lba,adr,p_lba,tlba
array raw8 buf_lo[256]              ;sector bytes   0..255
array raw8 buf_hi[256]              ;sector bytes 256..511

```

```
fail:=0
verr:=0
clearbit(LEDPORT,GREEN)
clearbit(LEDPORT,RED)

kind:=sd_init()
putc("I") putc("N") putc("I") putc("T") putc("Space")
if (kind=0) {
   fail:=1
   putc("F") putc("A") putc("I") putc("L")
}
altif (kind=1) { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("S") putc("C") }
default { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("H") putc("C") }
put_crlf()

take op from [0,6) {
   if (fail=0) {
      ; --- sector number of this operation -----------------------------------
      if (op=0) { lba:=0 }
      default { lba:=tlba }
      wr:=0
      if (op=2) { wr:=1 }
      if (op=4) { wr:=1 }

      ; --- what to write, op 2 = pattern, op 4 = zeros; op 3 clears the buffer --
      if (op=2) {
         k:=0
         loop {
            buf_lo[k]:=k
            t:=255-k
            buf_hi[k]:=t
            k:=k+1
            if (k=0) { quit }
         }
      }
      if (op>=3) {                          ;op 3, 4 (and 5): start from zeros
         if (op!=5) {
            k:=0
            loop {
               buf_lo[k]:=0
               buf_hi[k]:=0
               k:=k+1
               if (k=0) { quit }
            }
         }
      }

      ; --- sector number -> command argument (as in B4..B6) -------------------
      adr:=lba
      if (kind=1) { adr:=lba*512 }          ;SDSC = byte address
      a0:=raw8(adr)
      adr:=adr/256
      a1:=raw8(adr)
      adr:=adr/256
      a2:=raw8(adr)
      adr:=adr/256
      a3:=raw8(adr)

      if (wr=1) {
         ; --- the ONE write site = CMD24 ----------------------------------------
         wres:=255
         busy:=0
         clearbit(CSPORT,CS)
         r:=sd_cmd(88,a3,a2,a1,a0,255)      ;CMD24 = WRITE_BLOCK
         if (r=0) {
            t:=xchg(255)                    ;one byte gap before the token
            t:=xchg(254)                    ;start token FE
            k:=0
            loop {
               u:=buf_lo[k]
               t:=xchg(u)
               k:=k+1
               if (k=0) { quit }
            }
            loop {
               u:=buf_hi[k]
               t:=xchg(u)
               k:=k+1
               if (k=0) { quit }
            }
            t:=xchg(255)                    ;CRC16, not checked by the card
            t:=xchg(255)
            wres:=xchg(255)                 ;data response xxx0sss1
            wres:=wres&31                   ;05 = data accepted
            loop {                          ;busy, the card holds MISO low
               t:=xchg(255)
               if (t!=0) { quit }
```

```
                busy:=busy+1
                if (busy>=60000) { quit }
             }
          }
          sd_deselect()
          putc("W") putc("R") putc("I") putc("T") putc("E") putc("Space")
          putc("R") putc("Equal")
          put_hex(wres)
          putc("Space") putc("B") putc("U") putc("S") putc("Y") putc("Equal")
          t:=high(busy)
          put_hex(t)
          t:=low(busy)
          put_hex(t)
          put_crlf()
          if (wres!=5) { fail:=1 }
          if (busy>=60000) { fail:=1 }
       }
       default {
          ; --- the ONE read site = CMD17 (as in B3..B6) --------------------------
          mrk:=255
          clearbit(CSPORT,CS)
          r:=sd_cmd(81,a3,a2,a1,a0,255)
          if (r=0) {
             n:=0
             loop {
                mrk:=xchg(255)
                if (mrk!=255) { quit }
                n:=n+1
                if (n>=50000) { quit }
             }
          }
          if (mrk=254) {
             k:=0
             loop {
                t:=xchg(255)
                buf_lo[k]:=t
                k:=k+1
                if (k=0) { quit }
             }
             loop {
                t:=xchg(255)
                buf_hi[k]:=t
                k:=k+1
                if (k=0) { quit }
             }
             t:=xchg(255)                    ;CRC16, ignored
             t:=xchg(255)
          }
          sd_deselect()
          if (mrk!=254) {
             fail:=1
             putc("R") putc("E") putc("A") putc("D") putc("Space") putc("F") putc("A") putc("I
    ") putc("L")
             put_crlf()
          }
       }
    }

    ; --- evaluate ----------------------------------------------------------
    if (fail=0) {
       if (op=0) {                           ;MBR -> test sector in front of the partition
          ptype:=buf_hi[194]
          c3:=buf_hi[201]
          c2:=buf_hi[200]
          c1:=buf_hi[199]
          c0:=buf_hi[198]
          p_lba:=mk32(c3,c2,c1,c0)
          t:=buf_hi[254]
          u:=buf_hi[255]
          if (t!=85) { fail:=1 }             ;no MBR signature
          if (u!=170) { fail:=1 }
          if (ptype=0) { fail:=1 }           ;no partition 1
          if (p_lba<2) { fail:=1 }           ;no gap in front of the partition
          tlba:=p_lba-1
          putc("T") putc("E") putc("S") putc("T") putc("Space") putc("L") putc("B") putc("A")
      putc("Equal")
          put_hex32(tlba)
          if (fail!=0) { putc("Space") putc("F") putc("A") putc("I") putc("L") }
          put_crlf()
       }
       if (op=1) {                           ;the test sector must be empty
          errs:=0
          k:=0
          loop {
             t:=buf_lo[k]
             if (t!=0) { errs:=errs+1 }
             t:=buf_hi[k]
```

```
            if (t!=0) { errs:=errs+1 }
            k:=k+1
            if (k=0) { quit }
         }
         if (errs=0) { putc("Z") putc("E") putc("R") putc("O") putc("Space") putc("O") putc("
      K") }
         default {
            fail:=1                          ;not empty -> do NOT write
            putc("N") putc("O") putc("T") putc("Space") putc("E") putc("M") putc("P") putc("T
      ") putc("Y")
         }
         put_crlf()
       }
       if (op=3) {                           ;compare with the pattern
          errs:=0
          k:=0
          loop {
             t:=buf_lo[k]
             if (t!=k) { errs:=errs+1 }
             t:=buf_hi[k]
             u:=255-k
             if (t!=u) { errs:=errs+1 }
             k:=k+1
             if (k=0) { quit }
          }
          putc("V") putc("E") putc("R") putc("I") putc("F") putc("Y") putc("Space")
          putc("E") putc("R") putc("R") putc("Equal")
          t:=high(errs)
          put_hex(t)
          t:=low(errs)
          put_hex(t)
          put_crlf()
          if (errs!=0) { fail:=2 }            ;still restore the zeros (op 4)
       }
       if (op=5) {                           ;zeros again?
          errs:=0
          k:=0
          loop {
             t:=buf_lo[k]
             if (t!=0) { errs:=errs+1 }
             t:=buf_hi[k]
             if (t!=0) { errs:=errs+1 }
             k:=k+1
             if (k=0) { quit }
          }
          putc("R") putc("E") putc("S") putc("T") putc("O") putc("R") putc("E") putc("Space")
          if (errs=0) { putc("O") putc("K") }
          default {
             fail:=1
             putc("F") putc("A") putc("I") putc("L")
          }
          put_crlf()
       }
    }
    if (fail=2) {                            ;verify error, go on, the zeros must be restored
       fail:=0
       verr:=1                               ;remember the error for the end
    }
 }

 if (verr=1) { fail:=1 }                     ;the verify error counts in the end
 if (fail=0) { setbit(LEDPORT,GREEN) }
 default { setbit(LEDPORT,RED) }
 loop { delay(1000) }                        ;done, wait for reset
```

A.8 Test program `18_sd_8_append.jan`

```
; Program name: 18_sd_8_append.jan
; Object file name: 18_sd_8_append.asm
; Uses: ascii.jan
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 8 (append to a file)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card (FAT32, MBR) inserted.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 16/10/2026
; Version: 0.1
; Description: appends LINES copies of the text line txt (29 bytes, ends with CR LF)
;   to the file TEST02.TXT in the root directory and updates its size. When the last
;   cluster is full, a free cluster is taken from the FAT, marked as end of chain and
;   linked to the old last cluster, in BOTH FAT copies.
```

```
;   Each pass of the main loop does at most ONE sector access (io = 0 none, 1 read,
;   2 write); the state decides the sector number and what happens with the data.
;   state  io     sector                   action
;    0     read   0                        MBR -> p_lba
;    1     read   p_lba                    BPB -> spc, fat_lba, data_lba, cbytes, maxc
;    2     read   cluster c, sector s      search TEST02  TXT -> dlba, de, fc, fsize
;    3     read   FAT sector of c          next directory cluster
;    5     read   FAT sector of c          walk the file chain to its last cluster
;    6     none   -                        last cluster full? -> 7 (allocate) or 8
;    7     read   FAT sector fi            find a free entry -> newc, mark it end of chain
;   71     write  same sector, FAT 1
;   72     write  same sector, FAT 2       (skipped if NumFATs = 1)
;   73     read   FAT sector of c          link c -> newc
;   74     write  same sector, FAT 1
;   75     write  same sector, FAT 2
;   76     none   -                        newc becomes the last cluster, coff = 0
;    8     read   data sector of coff      read-modify (no read if it starts empty)
;    9     write  the same data sector     -> 8 (more bytes) / 6 (cluster full) / 10
;   10     read   directory sector dlba    new size (and first cluster if it was empty)
;   11     write  directory sector dlba    -> done
;   coff = bytes used in the last cluster (1..cbytes; cbytes = full).
;   Not updated (by design, see report) the FSInfo free-cluster hint and the
;   modification time stamp of the entry.
; Expected output (LINES = 1, TEST02.TXT of 71 bytes, 4 KB clusters)
;   INIT OK SDSC
;   FOUND C=00000005 S=00000047
;   LAST C=00000005 USED=00000047
;   WRITE LBA=00002098 R=05
;   WRITE LBA=00002080 R=05
;   SIZE=00000064 OK
; Each run appends again. To test the cluster allocation set LINES to 150
; (150 x 29 = 4350 bytes > the 4025 bytes still free in the first cluster).
; Runs once; green LED = done, red LED = error.
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;card deselected before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ---------------------------------------------------------------------------
; helpers (from blocks B2..B4)
; ---------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
   return(r)
}

; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                     ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                           ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }           ;no answer -> r stays FF
   }
   return(r)
}

; CS high, then 8 more clocks so the card releases MISO
subroutine sd_deselect() {
   raw8 r
```

```
   setbit(CSPORT,CS)
   r:=xchg(255)
}

subroutine putc(raw8 ch) {
   .uart_tx_byte(ch,context,local_id)
}

subroutine put_hex(raw8 v) {
   raw8 d,c
   d:=v/16
   c:=digit_to_ascii(d)
   putc(c)
   d:=v%16
   c:=digit_to_ascii(d)
   putc(c)
}

subroutine put_crlf() {
   putc("CR")
   putc("LF")
}

; (since B4) four bytes (most significant first) -> one raw32 value
function raw32 mk32(raw8 b3,raw8 b2,raw8 b1,raw8 b0) {
   raw32 v
   v:=raw32(b3)
   v:=v*256
   v:=v+raw32(b2)
   v:=v*256
   v:=v+raw32(b1)
   v:=v*256
   v:=v+raw32(b0)
   return(v)
}

; (since B4) raw32 -> 8 hex digits; raw8(w) keeps the low byte (narrowing), w/256 drops it
subroutine put_hex32(raw32 v) {
   raw32 w
   raw8 b3,b2,b1,b0
   w:=v
   b0:=raw8(w)
   w:=w/256
   b1:=raw8(w)
   w:=w/256
   b2:=raw8(w)
   w:=w/256
   b3:=raw8(w)
   put_hex(b3)
   put_hex(b2)
   put_hex(b1)
   put_hex(b0)
}

; ------------------------------------------------------------------------
; block B2 as one silent function (unchanged since B3).
; returns 0 = initialization failed, 1 = SDSC (byte addresses), 2 = SDHC (block addresses)
; ------------------------------------------------------------------------
function raw8 sd_init() {
   raw8 r,k,st,tries,v2,hcs,crc41,last,ocr0,kind
   st:=0
   kind:=0
   setbit(CSPORT,CS)
   .spi_set_speed(SPI_SLOW)
   take k from [0,10) { r:=xchg(255) }       ;80 clocks with CS high

   clearbit(CSPORT,CS)
   r:=sd_cmd(64,0,0,0,0,149)                 ;CMD0 -> expect 01
   sd_deselect()
   if (r!=1) { st:=1 }

   v2:=0
   if (st=0) {
      clearbit(CSPORT,CS)
      r:=sd_cmd(72,0,0,1,170,135)            ;CMD8 -> expect 01 + 000001AA
      take k from [0,4) { last:=xchg(255) } ;keep only the echo byte AA
      sd_deselect()
      if (r=1) {
         if (last=170) { v2:=1 }
         default { st:=1 }
      }
      altif (r=5) { }                        ;V1 card, no R7
      default { st:=1 }
   }

   if (st=0) {
      if (v2=1) { hcs:=64  crc41:=119 }
```

```
        default { hcs:=0  crc41:=229 }
        tries:=0
        loop {
           clearbit(CSPORT,CS)
           r:=sd_cmd(119,0,0,0,0,101)          ;CMD55
           r:=sd_cmd(105,hcs,0,0,0,crc41)      ;ACMD41
           sd_deselect()
           if (r=0) { quit }
           tries:=tries+1
           if (tries>=200) { quit }
           delay(10)
        }
        if (r!=0) { st:=1 }
     }

     if (st=0) {
        clearbit(CSPORT,CS)
        r:=sd_cmd(122,0,0,0,0,253)             ;CMD58 -> OCR
        ocr0:=xchg(255)                        ;first OCR byte holds CCS (bit 6)
        take k from [0,3) { last:=xchg(255) }
        sd_deselect()
        if (r!=0) { st:=1 }
     }

     if (st=0) {
        ocr0:=ocr0&64
        if (ocr0!=0) { kind:=2 }
        default { kind:=1 }
        .spi_set_speed(SPI_FAST)               ;4 MHz for the data transfer
     }
     return(kind)
  }

  ; ------------------------------------------------------------------------
  ; main
  ; ------------------------------------------------------------------------
  constant LINES:=1                  ;number of text lines appended per run
  constant LEN:=29                   ;bytes per line
  array raw8 txt[LEN]:=["M","y","B","u","g","Space","a","p","p","e","n","d","Space","t","e","s"
       ,"t","Space","P","I","C","_onE","_eighT","F","_fouR","_fivE","_twO","CR","LF"]
  array raw8 target[11]:=["T","E","S","T","_zerO","_twO","Space","Space","T","X","T"]

  raw8 r,k,t,u,mrk,fail,kind,state,io,wres,ptype,nf,i,a3,a2,a1,a0,c3,c2,c1,c0
  raw8 spc,s,e,j,half,base,idx,q,attr,match,de,ci,empty,found
  raw16 n,busy,bo,fi,nleft
  raw32 lba,adr,p_lba,rsvd,fsz,fat_lba,data_lba,c,pm,fsize,cbytes,coff,newc,dlba,maxc,tsec,
       firstnew,nclus
  array raw8 buf_lo[256]             ;sector bytes   0..255
  array raw8 buf_hi[256]             ;sector bytes 256..511
  array raw8 ent[32]                 ;copy of the current 32-byte directory entry

  fail:=0
  clearbit(LEDPORT,GREEN)
  clearbit(LEDPORT,RED)

  kind:=sd_init()
  putc("I") putc("N") putc("I") putc("T") putc("Space")
  if (kind=0) {
     fail:=1
     putc("F") putc("A") putc("I") putc("L")
  }
  altif (kind=1) { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("S") putc("C") }
  default { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("H") putc("C") }
  put_crlf()

  state:=0
  nleft:=LINES*LEN
  ci:=0
  empty:=0
  firstnew:=0
  loop {
     if (fail!=0) { quit }

     ; ====================================================================
     ; A. prepare = sector number and kind of access for this state
     ; ====================================================================
     io:=1                                 ;most states read
     if (state=0) { lba:=0 }
     altif (state=1) { lba:=p_lba }
     altif (state=3) {
        pm:=c/128
        lba:=fat_lba+pm
     }
     altif (state=5) {
        pm:=c/128
        lba:=fat_lba+pm
     }
```

```
277     altif (state=6) { io:=0 }
278     altif (state=7) { lba:=fat_lba+raw32(fi) }
279     altif (state=71) {
280        lba:=fat_lba+raw32(fi)
281        io:=2
282     }
283     altif (state=72) {
284        lba:=fat_lba+fsz
285        lba:=lba+raw32(fi)
286        io:=2
287     }
288     altif (state=73) {
289        pm:=c/128
290        lba:=fat_lba+pm
291     }
292     altif (state=74) {
293        pm:=c/128
294        lba:=fat_lba+pm
295        io:=2
296     }
297     altif (state=75) {
298        pm:=c/128
299        lba:=fat_lba+pm
300        lba:=lba+fsz
301        io:=2
302     }
303     altif (state=9) { io:=2 }                ;same data sector as state 8
304     altif (state=10) { lba:=dlba }
305     altif (state=11) {
306        lba:=dlba
307        io:=2
308     }
309     default { }                              ;states 2 and 8 below
310
311     if (state=8) {                           ;s = sector in the cluster, bo = byte in it
312        pm:=coff/512
313        s:=raw8(pm)
314        pm:=pm*512
315        pm:=coff-pm
316        bo:=raw16(pm)
317        if (bo=0) {                           ;sector starts empty -> zeros, no read
318           io:=0
319           k:=0
320           loop {
321              buf_lo[k]:=0
322              buf_hi[k]:=0
323              k:=k+1
324              if (k=0) { quit }
325           }
326        }
327     }
328     if (state=2) { io:=1 }
329     t:=0
330     if (state=2) { t:=1 }
331     if (state=8) { t:=1 }
332     if (t=1) {                               ;lba = data_lba + (c-2)*spc + s
333        pm:=c-2
334        lba:=0
335        i:=0
336        loop {
337           if (i>=spc) { quit }
338           lba:=lba+pm
339           i:=i+1
340        }
341        lba:=lba+data_lba
342        lba:=lba+raw32(s)
343     }
344
345     ; ====================================================================
346     ; B. the ONE sector access
347     ; ====================================================================
348     if (io!=0) {
349        adr:=lba
350        if (kind=1) { adr:=lba*512 }          ;SDSC = byte address
351        a0:=raw8(adr)
352        adr:=adr/256
353        a1:=raw8(adr)
354        adr:=adr/256
355        a2:=raw8(adr)
356        adr:=adr/256
357        a3:=raw8(adr)
358     }
359     if (io=1) {                              ;read site, CMD17
360        mrk:=255
361        clearbit(CSPORT,CS)
362        r:=sd_cmd(81,a3,a2,a1,a0,255)
363        if (r=0) {
```

```
364            n:=0
365            loop {
366               mrk:=xchg(255)
367               if (mrk!=255) { quit }
368               n:=n+1
369               if (n>=50000) { quit }
370            }
371         }
372         if (mrk=254) {
373            k:=0
374            loop {
375               t:=xchg(255)
376               buf_lo[k]:=t
377               k:=k+1
378               if (k=0) { quit }
379            }
380            loop {
381               t:=xchg(255)
382               buf_hi[k]:=t
383               k:=k+1
384               if (k=0) { quit }
385            }
386            t:=xchg(255)                        ;CRC16, ignored
387            t:=xchg(255)
388         }
389         sd_deselect()
390         if (mrk!=254) {
391            fail:=1
392            putc("R") putc("E") putc("A") putc("D") putc("Space") putc("F") putc("A") putc("I")
       putc("L")
393            put_crlf()
394            quit
395         }
396      }
397      if (io=2) {                               ;write site, CMD24
398         wres:=255
399         busy:=0
400         clearbit(CSPORT,CS)
401         r:=sd_cmd(88,a3,a2,a1,a0,255)
402         if (r=0) {
403            t:=xchg(255)                        ;one byte gap
404            t:=xchg(254)                        ;start token FE
405            k:=0
406            loop {
407               u:=buf_lo[k]
408               t:=xchg(u)
409               k:=k+1
410               if (k=0) { quit }
411            }
412            loop {
413               u:=buf_hi[k]
414               t:=xchg(u)
415               k:=k+1
416               if (k=0) { quit }
417            }
418            t:=xchg(255)                        ;CRC16, not checked
419            t:=xchg(255)
420            wres:=xchg(255)
421            wres:=wres&31                       ;05 = data accepted
422            loop {                              ;busy = MISO held low
423               t:=xchg(255)
424               if (t!=0) { quit }
425               busy:=busy+1
426               if (busy>=60000) { quit }
427            }
428         }
429         sd_deselect()
430         putc("W") putc("R") putc("I") putc("T") putc("E") putc("Space") putc("L") putc("B")
       putc("A") putc("Equal")
431         put_hex32(lba)
432         putc("Space") putc("R") putc("Equal")
433         put_hex(wres)
434         put_crlf()
435         if (wres!=5) { fail:=1 }
436         if (busy>=60000) { fail:=1 }
437         if (fail!=0) { quit }
438      }
439
440      ; ====================================================================
441      ; C. evaluate and choose the next state
442      ; ====================================================================
443      if (state=0) {                            ;MBR, partition entry 1
444         ptype:=buf_hi[194]
445         c3:=buf_hi[201]
446         c2:=buf_hi[200]
447         c1:=buf_hi[199]
448         c0:=buf_hi[198]
```

```
449         p_lba:=mk32(c3,c2,c1,c0)
450         if (ptype=11) { }
451         altif (ptype=12) { }
452         default { fail:=1 }
453         state:=1
454      }
455      altif (state=1) {                          ;boot sector, BPB
456         spc:=buf_lo[13]
457         nf:=buf_lo[16]
458         c1:=buf_lo[15]
459         c0:=buf_lo[14]
460         rsvd:=mk32(0,0,c1,c0)
461         c3:=buf_lo[39]
462         c2:=buf_lo[38]
463         c1:=buf_lo[37]
464         c0:=buf_lo[36]
465         fsz:=mk32(c3,c2,c1,c0)
466         c3:=buf_lo[35]                          ;TotSec32 = sectors of the partition
467         c2:=buf_lo[34]
468         c1:=buf_lo[33]
469         c0:=buf_lo[32]
470         tsec:=mk32(c3,c2,c1,c0)
471         fat_lba:=p_lba+rsvd
472         data_lba:=fat_lba
473         i:=0
474         loop {
475            if (i>=nf) { quit }
476            data_lba:=data_lba+fsz
477            i:=i+1
478         }
479         cbytes:=raw32(spc)*512                  ;bytes per cluster
480         maxc:=data_lba-p_lba                    ;sectors in front of the data area
481         maxc:=tsec-maxc                         ;sectors of the data area
482         pm:=raw32(spc)
483         loop {                                  ;divide by spc (a power of two) by halving
484            if (pm<2) { quit }
485            maxc:=maxc/2
486            pm:=pm/2
487         }
488         maxc:=maxc+1                            ;highest valid cluster number
489         c3:=buf_lo[47]                          ;RootClus
490         c2:=buf_lo[46]
491         c1:=buf_lo[45]
492         c0:=buf_lo[44]
493         c:=mk32(c3,c2,c1,c0)
494         s:=0
495         if (spc=0) { fail:=1 }
496         if (nf=0) { fail:=1 }
497         if (nf>2) { fail:=1 }
498         state:=2
499      }
500      altif (state=2) {                          ;directory sector, search target
501         found:=0
502         e:=0
503         loop {
504            half:=e/8
505            base:=e%8
506            base:=base*32
507            take j from [0,32) {
508               idx:=base+j
509               if (half=0) { t:=buf_lo[idx] }
510               default { t:=buf_hi[idx] }
511               ent[j]:=t
512            }
513            t:=ent[0]
514            if (t=0) {                           ;end of directory
515               putc("N") putc("O") putc("T") putc("Space") putc("F") putc("O") putc("U") putc("N
     ") putc("D")
516               put_crlf()
517               fail:=1
518               quit
519            }
520            match:=1
521            take j from [0,11) {
522               t:=ent[j]
523               u:=target[j]
524               if (t!=u) { match:=0 }
525            }
526            attr:=ent[11]
527            if (attr=15) { match:=0 }
528            t:=attr&16
529            if (t!=0) { match:=0 }
530            if (match=1) {
531               found:=1
532               dlba:=lba                         ;remember where the entry lives
533               de:=e
534               c3:=ent[21]
```

```
            c2:=ent[20]
            c1:=ent[27]
            c0:=ent[26]
            c:=mk32(c3,c2,c1,c0)
            c3:=ent[31]
            c2:=ent[30]
            c1:=ent[29]
            c0:=ent[28]
            fsize:=mk32(c3,c2,c1,c0)
            putc("F") putc("O") putc("U") putc("N") putc("D") putc("Space") putc("C") putc("
   Equal")
            put_hex32(c)
            putc("Space") putc("S") putc("Equal")
            put_hex32(fsize)
            put_crlf()
            quit
         }
         e:=e+1
         if (e>=16) { quit }
      }
      if (found=1) {
         if (c=0) {                          ;empty file, no cluster yet
            empty:=1
            coff:=cbytes                     ;behaves like a full cluster -> allocate
            state:=6
         }
         default {
            nclus:=1
            state:=5
         }
      }
      altif (fail=0) {                       ;not in this sector
         s:=s+1
         if (s>=spc) { state:=3 }
      }
      default { }
   }
   altif (state=3) {                         ;next cluster of the directory
      q:=raw8(c)
      q:=q&127
      half:=q/64
      idx:=q&63
      idx:=idx*4
      if (half=0) {
         c0:=buf_lo[idx]
         idx:=idx+1
         c1:=buf_lo[idx]
         idx:=idx+1
         c2:=buf_lo[idx]
         idx:=idx+1
         c3:=buf_lo[idx]
      }
      default {
         c0:=buf_hi[idx]
         idx:=idx+1
         c1:=buf_hi[idx]
         idx:=idx+1
         c2:=buf_hi[idx]
         idx:=idx+1
         c3:=buf_hi[idx]
      }
      c3:=c3&15
      c:=mk32(c3,c2,c1,c0)
      if (c>=268435448) { fail:=1 }          ;end of the directory, file not found
      altif (c<2) { fail:=1 }
      default {
         s:=0
         state:=2
      }
   }
   altif (state=5) {                         ;walk the file chain to the last cluster
      q:=raw8(c)
      q:=q&127
      half:=q/64
      idx:=q&63
      idx:=idx*4
      if (half=0) {
         c0:=buf_lo[idx]
         idx:=idx+1
         c1:=buf_lo[idx]
         idx:=idx+1
         c2:=buf_lo[idx]
         idx:=idx+1
         c3:=buf_lo[idx]
      }
      default {
         c0:=buf_hi[idx]
```

```
            idx:=idx+1
            c1:=buf_hi[idx]
            idx:=idx+1
            c2:=buf_hi[idx]
            idx:=idx+1
            c3:=buf_hi[idx]
         }
         c3:=c3&15
         pm:=mk32(c3,c2,c1,c0)                  ;FAT[c]
         if (pm>=268435448) {                   ;c is the last cluster
            coff:=0                             ;coff = fsize - (nclus-1)*cbytes
            pm:=1
            loop {                              ;(nclus-1)*cbytes by repeated addition
               if (pm>=nclus) { quit }
               coff:=coff+cbytes
               pm:=pm+1
            }
            coff:=fsize-coff
            putc("L") putc("A") putc("S") putc("T") putc("Space") putc("C") putc("Equal")
            put_hex32(c)
            putc("Space") putc("U") putc("S") putc("E") putc("D") putc("Equal")
            put_hex32(coff)
            put_crlf()
            if (coff>cbytes) { fail:=1 }        ;size and chain disagree
            state:=6
         }
         altif (pm<2) { fail:=1 }
         default {
            c:=pm
            nclus:=nclus+1
            if (nclus>=60000) { fail:=1 }
         }
      }
      altif (state=6) {                         ;room left in the last cluster?
         if (coff>=cbytes) {
            fi:=0
            state:=7
         }
         default { state:=8 }
      }
      altif (state=7) {                         ;search a free FAT entry in sector fi
         found:=0
         j:=0
         loop {
            newc:=raw32(fi)*128
            newc:=newc+raw32(j)
            half:=j/64
            idx:=j&63
            idx:=idx*4
            if (half=0) {
               c0:=buf_lo[idx]
               idx:=idx+1
               c1:=buf_lo[idx]
               idx:=idx+1
               c2:=buf_lo[idx]
               idx:=idx+1
               c3:=buf_lo[idx]
            }
            default {
               c0:=buf_hi[idx]
               idx:=idx+1
               c1:=buf_hi[idx]
               idx:=idx+1
               c2:=buf_hi[idx]
               idx:=idx+1
               c3:=buf_hi[idx]
            }
            c3:=c3&15
            t:=0                                ;free entry = all 28 bits zero
            if (c0!=0) { t:=1 }
            if (c1!=0) { t:=1 }
            if (c2!=0) { t:=1 }
            if (c3!=0) { t:=1 }
            if (newc<2) { t:=1 }                ;clusters 0 and 1 are reserved
            if (newc>maxc) { t:=1 }             ;beyond the end of the data area
            if (t=0) {
               found:=1
               quit
            }
            j:=j+1
            if (j>=128) { quit }
         }
         if (found=1) {                         ;mark newc as end of chain (0FFFFFFF)
            half:=j/64                          ;idx points to byte 3 of the entry
            if (half=0) {
               t:=buf_lo[idx]
               t:=t&240                         ;keep the 4 reserved bits
```

```
            t:=t+15                         ;(low nibble is 0 now, + acts as OR)
            buf_lo[idx]:=t
            idx:=idx-1
            buf_lo[idx]:=255
            idx:=idx-1
            buf_lo[idx]:=255
            idx:=idx-1
            buf_lo[idx]:=255
         }
         default {
            t:=buf_hi[idx]
            t:=t&240
            t:=t+15
            buf_hi[idx]:=t
            idx:=idx-1
            buf_hi[idx]:=255
            idx:=idx-1
            buf_hi[idx]:=255
            idx:=idx-1
            buf_hi[idx]:=255
         }
         putc("N") putc("E") putc("W") putc("Space") putc("C") putc("Equal")
         put_hex32(newc)
         put_crlf()
         state:=71
      }
      default {
         fi:=fi+1
         pm:=raw32(fi)
         if (pm>=fsz) {                     ;no free cluster at all
            putc("F") putc("U") putc("L") putc("L")
            put_crlf()
            fail:=1
         }
      }
   }
   altif (state=71) {                       ;FAT 1 written
      if (nf=2) { state:=72 }
      altif (empty=1) { state:=76 }
      default { state:=73 }
   }
   altif (state=72) {                       ;FAT 2 written
      if (empty=1) { state:=76 }
      default { state:=73 }
   }
   altif (state=73) {                       ;FAT[c] := newc (link the old last cluster)
      q:=raw8(c)
      q:=q&127
      half:=q/64
      idx:=q&63
      idx:=idx*4
      pm:=newc
      c0:=raw8(pm)
      pm:=pm/256
      c1:=raw8(pm)
      pm:=pm/256
      c2:=raw8(pm)
      pm:=pm/256
      c3:=raw8(pm)
      c3:=c3&15
      if (half=0) {
         buf_lo[idx]:=c0
         idx:=idx+1
         buf_lo[idx]:=c1
         idx:=idx+1
         buf_lo[idx]:=c2
         idx:=idx+1
         t:=buf_lo[idx]
         t:=t&240
         t:=t+c3                            ;c3 <= 15, + acts as OR
         buf_lo[idx]:=t
      }
      default {
         buf_hi[idx]:=c0
         idx:=idx+1
         buf_hi[idx]:=c1
         idx:=idx+1
         buf_hi[idx]:=c2
         idx:=idx+1
         t:=buf_hi[idx]
         t:=t&240
         t:=t+c3
         buf_hi[idx]:=t
      }
      state:=74
   }
   altif (state=74) {                       ;FAT 1 written
```

```
        if (nf=2) { state:=75 }
        default { state:=76 }
    }
    altif (state=75) { state:=76 }            ;FAT 2 written
    altif (state=8) {                         ;put the next bytes into the sector
        loop {
            if (nleft=0) { quit }
            if (bo>=512) { quit }
            t:=txt[ci]
            half:=high(bo)
            idx:=low(bo)
            if (half=0) { buf_lo[idx]:=t }
            default { buf_hi[idx]:=t }
            ci:=ci+1
            if (ci>=LEN) { ci:=0 }
            bo:=bo+1
            coff:=coff+1
            fsize:=fsize+1
            nleft:=nleft-1
        }
        state:=9
    }
    altif (state=9) {                         ;data sector written
        if (nleft=0) { state:=10 }
        altif (coff>=cbytes) { state:=6 }
        default { state:=8 }
    }
    altif (state=10) {                        ;directory entry, new size (+ first cluster)
        half:=de/8
        base:=de%8
        base:=base*32
        pm:=fsize
        take j from [28,32) {                 ;size, little-endian, offset 28..31
            idx:=base+j
            t:=raw8(pm)
            if (half=0) { buf_lo[idx]:=t }
            default { buf_hi[idx]:=t }
            pm:=pm/256
        }
        if (empty=2) {                        ;first cluster, low word 26/27, high 20/21
            pm:=firstnew
            take j from [0,4) {
                if (j=0) { idx:=base+26 }
                altif (j=1) { idx:=base+27 }
                altif (j=2) { idx:=base+20 }
                default { idx:=base+21 }
                t:=raw8(pm)
                if (half=0) { buf_lo[idx]:=t }
                default { buf_hi[idx]:=t }
                pm:=pm/256
            }
        }
        state:=11
    }
    altif (state=11) {                        ;directory sector written
        putc("S") putc("I") putc("Z") putc("E") putc("Equal")
        put_hex32(fsize)
        putc("Space") putc("O") putc("K")
        put_crlf()
        state:=99
    }
    default { fail:=1 }                       ;unknown state

    ; state 76 (internal) = the new cluster becomes the last one
    if (state=76) {
        if (empty=1) {
            firstnew:=newc
            empty:=2                          ;entry still needs the first cluster
        }
        c:=newc
        coff:=0
        state:=8
    }
    if (state=99) { quit }
}

if (fail=0) { setbit(LEDPORT,GREEN) }
default { setbit(LEDPORT,RED) }
loop { delay(1000) }                          ;done, wait for reset
```

A.9 Test program `18_sd_9_create.jan`

```
; Program name: 18_sd_9_create.jan
; Object file name: 18_sd_9_create.asm
```

```
; Uses: ascii.jan
; MCU: PIC18F452, HS crystal 16 MHz (adapt FREQUENCY to your crystal)
; Project: FAT32 + SD card on the PIC18F452, Baustein 9 (create a file)
; Configuration: no interrupts; used pins RA1 (red LED), RA0 (green LED),
;                RC3 (SCK), RC4 (SDI = MISO), RC5 (SDO = MOSI), RC2 (CS),
;                RC6 (TX), RC7 (RX); USART asynchronous 9600 Bd
; Hardware: AZ-Delivery micro-SD module on 5 V, SD/SDHC card (FAT32, MBR) inserted.
; Author: Claude Baumann
; License: CC BY-NC-SA 4.0 (attribution, non-commercial, share alike; see the report)
; Date: 16/10/2026
; Version: 0.1
; Description: creates the new file MYBUG01.TXT in the root directory and writes
;   LINES copies of the line txt ("Created by MyBug on a PIC18F452." CR LF, 34 bytes).
;   Built on block B8 (same three-part state machine, same allocation and write code)
;   state 0, 1    MBR, BPB (as in B8)
;   state 2       directory sector, the name must NOT exist yet ("EXISTS" -> stop);
;                 the first free slot (first byte E5 = deleted, or 00 = end) is kept
;                 in fdlba/fde; at 00 the search is complete -> state 20
;   state 3       next directory cluster; at the end of the chain a deleted slot is
;                 used, else "DIR FULL" (extending the directory is not implemented)
;   state 20      (no access) new, empty file -> coff = cbytes -> state 6
;   state 6, 7, 71, 72, 76   allocate the first cluster (as in B8, no link needed)
;   state 8, 9    write the text into the new cluster (sectors start empty, no read)
;   state 10, 11  read the directory sector, write the complete 32-byte entry
;                 name, attribute 20, times 12h00m00s, dates 2026-10-16 (the PIC has
;                 no clock yet), first cluster, size -- the entry is written LAST
; Expected output (LINES = 3 -> 102 bytes)
;   INIT OK SDSC
;   FREE LBA=00002080 E=xx
;   NEW C=0000xxxx
;   WRITE LBA=00000328 R=05
;   WRITE LBA=000011D4 R=05
;   WRITE LBA=0000xxxx R=05
;   WRITE LBA=00002080 R=05
;   SIZE=00000066 OK
; A second run stops with EXISTS (delete MYBUG01.TXT on the PC to repeat).
; Runs once; green LED = done, red LED = error.
;------------------------------------------------------------

<insert> _specific_pic18F452.jan </insert>
<insert> _pic18Fxx_.jan </insert>
<insert> iCode_Microchip_18F452.jan </insert>
<insert> ascii.jan </insert>
<define> FREQUENCY:=16000000 </define>

<define> LEDPORT:=3977 </define>   ;LATA 0xF89 (own name, do not shadow the SFR LATx)
<define> RED:=1 </define>          ;RA1
<define> GREEN:=0 </define>        ;RA0
<define> CSPORT:=3979 </define>    ;LATC 0xF8B
<define> CS:=2 </define>           ;RC2 = SD chip select (active low)

.configure(HS,WDT_OFF,BOR_OFF,NONE,MJ_OFF)
setbit(CSPORT,CS)                  ;card deselected before SPI starts
.configure_uart(9600,NONE)
.configure_spi(SPI_SLOW)
use_ascii_functions()

; ---------------------------------------------------------------------------
; helpers (from blocks B2..B4)
; ---------------------------------------------------------------------------
; one SPI byte exchange (send b, return the byte received meanwhile)
function raw8 xchg(raw8 b) {
   raw8 r
   .spi_xfer(b,r,context,local_id)
   return(r)
}

; send one SD command frame (6 bytes) and return its R1 answer.
; cmd = 64 + command index; a3..a0 = 32-bit argument, MSB first; crc = CRC7 + end bit
function raw8 sd_cmd(raw8 cmd,raw8 a3,raw8 a2,raw8 a1,raw8 a0,raw8 crc) {
   raw8 r,n
   r:=xchg(255)                     ;one idle byte = 8 clocks, card gets ready
   r:=xchg(cmd)
   r:=xchg(a3)
   r:=xchg(a2)
   r:=xchg(a1)
   r:=xchg(a0)
   r:=xchg(crc)
   n:=0
   loop {                           ;R1 comes after 1..8 bytes (NCR), bit 7 = 0
      r:=xchg(255)
      if (r<128) { quit }
      n:=n+1
      if (n>=10) { quit }           ;no answer -> r stays FF
   }
   return(r)
}
```

```
90
91  ; CS high, then 8 more clocks so the card releases MISO
92  subroutine sd_deselect() {
93     raw8 r
94     setbit(CSPORT,CS)
95     r:=xchg(255)
96  }
97
98  subroutine putc(raw8 ch) {
99     .uart_tx_byte(ch,context,local_id)
100 }
101
102 subroutine put_hex(raw8 v) {
103    raw8 d,c
104    d:=v/16
105    c:=digit_to_ascii(d)
106    putc(c)
107    d:=v%16
108    c:=digit_to_ascii(d)
109    putc(c)
110 }
111
112 subroutine put_crlf() {
113    putc("CR")
114    putc("LF")
115 }
116
117 ; (since B4) four bytes (most significant first) -> one raw32 value
118 function raw32 mk32(raw8 b3,raw8 b2,raw8 b1,raw8 b0) {
119    raw32 v
120    v:=raw32(b3)
121    v:=v*256
122    v:=v+raw32(b2)
123    v:=v*256
124    v:=v+raw32(b1)
125    v:=v*256
126    v:=v+raw32(b0)
127    return(v)
128 }
129
130 ; (since B4) raw32 -> 8 hex digits; raw8(w) keeps the low byte (narrowing), w/256 drops it
131 subroutine put_hex32(raw32 v) {
132    raw32 w
133    raw8 b3,b2,b1,b0
134    w:=v
135    b0:=raw8(w)
136    w:=w/256
137    b1:=raw8(w)
138    w:=w/256
139    b2:=raw8(w)
140    w:=w/256
141    b3:=raw8(w)
142    put_hex(b3)
143    put_hex(b2)
144    put_hex(b1)
145    put_hex(b0)
146 }
147
148 ; ------------------------------------------------------------------------
149 ; block B2 as one silent function (unchanged since B3).
150 ; returns 0 = initialization failed, 1 = SDSC (byte addresses), 2 = SDHC (block addresses)
151 ; ------------------------------------------------------------------------
152 function raw8 sd_init() {
153    raw8 r,k,st,tries,v2,hcs,crc41,last,ocr0,kind
154    st:=0
155    kind:=0
156    setbit(CSPORT,CS)
157    .spi_set_speed(SPI_SLOW)
158    take k from [0,10) { r:=xchg(255) }       ;80 clocks with CS high
159
160    clearbit(CSPORT,CS)
161    r:=sd_cmd(64,0,0,0,0,149)                 ;CMD0 -> expect 01
162    sd_deselect()
163    if (r!=1) { st:=1 }
164
165    v2:=0
166    if (st=0) {
167       clearbit(CSPORT,CS)
168       r:=sd_cmd(72,0,0,1,170,135)            ;CMD8 -> expect 01 + 000001AA
169       take k from [0,4) { last:=xchg(255) } ;keep only the echo byte AA
170       sd_deselect()
171       if (r=1) {
172          if (last=170) { v2:=1 }
173          default { st:=1 }
174       }
175       altif (r=5) { }                        ;V1 card, no R7
176       default { st:=1 }
```

```
    }

    if (st=0) {
       if (v2=1) { hcs:=64  crc41:=119 }
       default { hcs:=0  crc41:=229 }
       tries:=0
       loop {
          clearbit(CSPORT,CS)
          r:=sd_cmd(119,0,0,0,0,101)          ;CMD55
          r:=sd_cmd(105,hcs,0,0,0,crc41)      ;ACMD41
          sd_deselect()
          if (r=0) { quit }
          tries:=tries+1
          if (tries>=200) { quit }
          delay(10)
       }
       if (r!=0) { st:=1 }
    }

    if (st=0) {
       clearbit(CSPORT,CS)
       r:=sd_cmd(122,0,0,0,0,253)             ;CMD58 -> OCR
       ocr0:=xchg(255)                        ;first OCR byte holds CCS (bit 6)
       take k from [0,3) { last:=xchg(255) }
       sd_deselect()
       if (r!=0) { st:=1 }
    }

    if (st=0) {
       ocr0:=ocr0&64
       if (ocr0!=0) { kind:=2 }
       default { kind:=1 }
       .spi_set_speed(SPI_FAST)               ;4 MHz for the data transfer
    }
    return(kind)
}

; ------------------------------------------------------------------------
; main
; ------------------------------------------------------------------------
constant LINES:=3                   ;number of text lines written into the new file
constant LEN:=34                    ;bytes per line
array raw8 txt[LEN]:=["C","r","e","a","t","e","d","Space","b","y","Space","M","y","B","u","g"
     ,"Space","o","n","Space","a","Space","P","I","C","_onE","_eighT","F","_fouR","_fivE","
     _twO","Dot","CR","LF"]
array raw8 target[11]:=["M","Y","B","U","G","_zerO","_onE","Space","T","X","T"]

raw8 r,k,t,u,mrk,fail,kind,state,io,wres,ptype,nf,i,a3,a2,a1,a0,c3,c2,c1,c0
raw8 spc,s,e,j,half,base,idx,q,attr,match,de,ci,empty,found,hasfree,fde
raw32 fdlba
raw16 n,busy,bo,fi,nleft
raw32 lba,adr,p_lba,rsvd,fsz,fat_lba,data_lba,c,pm,fsize,cbytes,coff,newc,dlba,maxc,tsec,
     firstnew
array raw8 buf_lo[256]              ;sector bytes   0..255
array raw8 buf_hi[256]              ;sector bytes 256..511
array raw8 ent[32]                  ;copy of the current 32-byte directory entry

fail:=0
clearbit(LEDPORT,GREEN)
clearbit(LEDPORT,RED)

kind:=sd_init()
putc("I") putc("N") putc("I") putc("T") putc("Space")
if (kind=0) {
   fail:=1
   putc("F") putc("A") putc("I") putc("L")
}
altif (kind=1) { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("S") putc("C") }
default { putc("O") putc("K") putc("Space") putc("S") putc("D") putc("H") putc("C") }
put_crlf()

state:=0
nleft:=LINES*LEN
ci:=0
empty:=0
firstnew:=0
hasfree:=0
fdlba:=0
fde:=0
loop {
   if (fail!=0) { quit }

   ; ====================================================================
   ; A. prepare = sector number and kind of access for this state
   ; ====================================================================
   io:=1                                ;most states read
   if (state=0) { lba:=0 }
```

```
    altif (state=1) { lba:=p_lba }
    altif (state=3) {
       pm:=c/128
       lba:=fat_lba+pm
    }
    altif (state=6) { io:=0 }
    altif (state=7) { lba:=fat_lba+raw32(fi) }
    altif (state=71) {
       lba:=fat_lba+raw32(fi)
       io:=2
    }
    altif (state=72) {
       lba:=fat_lba+fsz
       lba:=lba+raw32(fi)
       io:=2
    }
    altif (state=73) {
       pm:=c/128
       lba:=fat_lba+pm
    }
    altif (state=74) {
       pm:=c/128
       lba:=fat_lba+pm
       io:=2
    }
    altif (state=75) {
       pm:=c/128
       lba:=fat_lba+pm
       lba:=lba+fsz
       io:=2
    }
    altif (state=9) { io:=2 }                ;same data sector as state 8
    altif (state=10) { lba:=dlba }
    altif (state=11) {
       lba:=dlba
       io:=2
    }
    default { }                              ;states 2 and 8 below

    if (state=8) {                           ;s = sector in the cluster, bo = byte in it
       pm:=coff/512
       s:=raw8(pm)
       pm:=pm*512
       pm:=coff-pm
       bo:=raw16(pm)
       if (bo=0) {                           ;sector starts empty -> zeros, no read
          io:=0
          k:=0
          loop {
             buf_lo[k]:=0
             buf_hi[k]:=0
             k:=k+1
             if (k=0) { quit }
          }
       }
    }
    if (state=2) { io:=1 }
    t:=0
    if (state=2) { t:=1 }
    if (state=8) { t:=1 }
    if (t=1) {                               ;lba = data_lba + (c-2)*spc + s
       pm:=c-2
       lba:=0
       i:=0
       loop {
          if (i>=spc) { quit }
          lba:=lba+pm
          i:=i+1
       }
       lba:=lba+data_lba
       lba:=lba+raw32(s)
    }

    ; ====================================================================
    ; B. the ONE sector access
    ; ====================================================================
    if (io!=0) {
       adr:=lba
       if (kind=1) { adr:=lba*512 }          ;SDSC = byte address
       a0:=raw8(adr)
       adr:=adr/256
       a1:=raw8(adr)
       adr:=adr/256
       a2:=raw8(adr)
       adr:=adr/256
       a3:=raw8(adr)
    }
```

```
348     if (io=1) {                             ;read site, CMD17
349        mrk:=255
350        clearbit(CSPORT,CS)
351        r:=sd_cmd(81,a3,a2,a1,a0,255)
352        if (r=0) {
353           n:=0
354           loop {
355              mrk:=xchg(255)
356              if (mrk!=255) { quit }
357              n:=n+1
358              if (n>=50000) { quit }
359           }
360        }
361        if (mrk=254) {
362           k:=0
363           loop {
364              t:=xchg(255)
365              buf_lo[k]:=t
366              k:=k+1
367              if (k=0) { quit }
368           }
369           loop {
370              t:=xchg(255)
371              buf_hi[k]:=t
372              k:=k+1
373              if (k=0) { quit }
374           }
375           t:=xchg(255)                      ;CRC16, ignored
376           t:=xchg(255)
377        }
378        sd_deselect()
379        if (mrk!=254) {
380           fail:=1
381           putc("R") putc("E") putc("A") putc("D") putc("Space") putc("F") putc("A") putc("I")
        putc("L")
382           put_crlf()
383           quit
384        }
385     }
386     if (io=2) {                             ;write site, CMD24
387        wres:=255
388        busy:=0
389        clearbit(CSPORT,CS)
390        r:=sd_cmd(88,a3,a2,a1,a0,255)
391        if (r=0) {
392           t:=xchg(255)                      ;one byte gap
393           t:=xchg(254)                      ;start token FE
394           k:=0
395           loop {
396              u:=buf_lo[k]
397              t:=xchg(u)
398              k:=k+1
399              if (k=0) { quit }
400           }
401           loop {
402              u:=buf_hi[k]
403              t:=xchg(u)
404              k:=k+1
405              if (k=0) { quit }
406           }
407           t:=xchg(255)                      ;CRC16, not checked
408           t:=xchg(255)
409           wres:=xchg(255)
410           wres:=wres&31                     ;05 = data accepted
411           loop {                            ;busy = MISO held low
412              t:=xchg(255)
413              if (t!=0) { quit }
414              busy:=busy+1
415              if (busy>=60000) { quit }
416           }
417        }
418        sd_deselect()
419        putc("W") putc("R") putc("I") putc("T") putc("E") putc("Space") putc("L") putc("B")
        putc("A") putc("Equal")
420        put_hex32(lba)
421        putc("Space") putc("R") putc("Equal")
422        put_hex(wres)
423        put_crlf()
424        if (wres!=5) { fail:=1 }
425        if (busy>=60000) { fail:=1 }
426        if (fail!=0) { quit }
427     }
428
429     ; ====================================================================
430     ; C. evaluate and choose the next state
431     ; ====================================================================
432     if (state=0) {                          ;MBR, partition entry 1
```

```
433         ptype:=buf_hi[194]
434         c3:=buf_hi[201]
435         c2:=buf_hi[200]
436         c1:=buf_hi[199]
437         c0:=buf_hi[198]
438         p_lba:=mk32(c3,c2,c1,c0)
439         if (ptype=11) { }
440         altif (ptype=12) { }
441         default { fail:=1 }
442         state:=1
443      }
444      altif (state=1) {                          ;boot sector, BPB
445         spc:=buf_lo[13]
446         nf:=buf_lo[16]
447         c1:=buf_lo[15]
448         c0:=buf_lo[14]
449         rsvd:=mk32(0,0,c1,c0)
450         c3:=buf_lo[39]
451         c2:=buf_lo[38]
452         c1:=buf_lo[37]
453         c0:=buf_lo[36]
454         fsz:=mk32(c3,c2,c1,c0)
455         c3:=buf_lo[35]                          ;TotSec32 = sectors of the partition
456         c2:=buf_lo[34]
457         c1:=buf_lo[33]
458         c0:=buf_lo[32]
459         tsec:=mk32(c3,c2,c1,c0)
460         fat_lba:=p_lba+rsvd
461         data_lba:=fat_lba
462         i:=0
463         loop {
464            if (i>=nf) { quit }
465            data_lba:=data_lba+fsz
466            i:=i+1
467         }
468         cbytes:=raw32(spc)*512                  ;bytes per cluster
469         maxc:=data_lba-p_lba                    ;sectors in front of the data area
470         maxc:=tsec-maxc                         ;sectors of the data area
471         pm:=raw32(spc)
472         loop {                                  ;divide by spc (a power of two) by halving
473            if (pm<2) { quit }
474            maxc:=maxc/2
475            pm:=pm/2
476         }
477         maxc:=maxc+1                            ;highest valid cluster number
478         c3:=buf_lo[47]                          ;RootClus
479         c2:=buf_lo[46]
480         c1:=buf_lo[45]
481         c0:=buf_lo[44]
482         c:=mk32(c3,c2,c1,c0)
483         s:=0
484         if (spc=0) { fail:=1 }
485         if (nf=0) { fail:=1 }
486         if (nf>2) { fail:=1 }
487         state:=2
488      }
489      altif (state=2) {                          ;directory sector, search target
490         found:=0
491         e:=0
492         loop {
493            half:=e/8
494            base:=e%8
495            base:=base*32
496            take j from [0,32) {
497               idx:=base+j
498               if (half=0) { t:=buf_lo[idx] }
499               default { t:=buf_hi[idx] }
500               ent[j]:=t
501            }
502            t:=ent[0]
503            if (t=0) {                           ;00 = end of directory, this slot is free
504               if (hasfree=0) {
505                  hasfree:=1
506                  fdlba:=lba
507                  fde:=e
508               }
509               found:=1                          ;whole directory seen, name not present
510               quit
511            }
512            if (t=229) {                         ;E5 = deleted entry, may be reused
513               if (hasfree=0) {
514                  hasfree:=1
515                  fdlba:=lba
516                  fde:=e
517               }
518            }
519            default {                            ;used entry, the name must not exist yet
```

```
            match:=1
            take j from [0,11) {
               t:=ent[j]
               u:=target[j]
               if (t!=u) { match:=0 }
            }
            attr:=ent[11]
            if (attr=15) { match:=0 }        ;long-name pieces never count
            if (match=1) {
               putc("E") putc("X") putc("I") putc("S") putc("T") putc("S")
               put_crlf()
               fail:=1
               quit
            }
         }
         e:=e+1
         if (e>=16) { quit }
      }
      if (found=1) { state:=20 }             ;create the file in the free slot
      altif (fail=0) {                       ;go on with the next directory sector
         s:=s+1
         if (s>=spc) { state:=3 }
      }
      default { }
   }
   altif (state=3) {                         ;next cluster of the directory
      q:=raw8(c)
      q:=q&127
      half:=q/64
      idx:=q&63
      idx:=idx*4
      if (half=0) {
         c0:=buf_lo[idx]
         idx:=idx+1
         c1:=buf_lo[idx]
         idx:=idx+1
         c2:=buf_lo[idx]
         idx:=idx+1
         c3:=buf_lo[idx]
      }
      default {
         c0:=buf_hi[idx]
         idx:=idx+1
         c1:=buf_hi[idx]
         idx:=idx+1
         c2:=buf_hi[idx]
         idx:=idx+1
         c3:=buf_hi[idx]
      }
      c3:=c3&15
      c:=mk32(c3,c2,c1,c0)
      if (c>=268435448) {                    ;end of the directory chain without 00
         if (hasfree=1) { state:=20 }        ;a deleted slot can be reused
         default {
            putc("D") putc("I") putc("R") putc("Space") putc("F") putc("U") putc("L") putc("L
   ")
            put_crlf()
            fail:=1                          ;(extending the directory is not implemented)
         }
      }
      altif (c<2) { fail:=1 }
      default {
         s:=0
         state:=2
      }
   }
   altif (state=6) {                         ;room left in the last cluster?
      if (coff>=cbytes) {
         fi:=0
         state:=7
      }
      default { state:=8 }
   }
   altif (state=7) {                         ;search a free FAT entry in sector fi
      found:=0
      j:=0
      loop {
         newc:=raw32(fi)*128
         newc:=newc+raw32(j)
         half:=j/64
         idx:=j&63
         idx:=idx*4
         if (half=0) {
            c0:=buf_lo[idx]
            idx:=idx+1
            c1:=buf_lo[idx]
            idx:=idx+1
```

```
            c2:=buf_lo[idx]
            idx:=idx+1
            c3:=buf_lo[idx]
         }
         default {
            c0:=buf_hi[idx]
            idx:=idx+1
            c1:=buf_hi[idx]
            idx:=idx+1
            c2:=buf_hi[idx]
            idx:=idx+1
            c3:=buf_hi[idx]
         }
         c3:=c3&15
         t:=0                                ;free entry = all 28 bits zero
         if (c0!=0) { t:=1 }
         if (c1!=0) { t:=1 }
         if (c2!=0) { t:=1 }
         if (c3!=0) { t:=1 }
         if (newc<2) { t:=1 }                ;clusters 0 and 1 are reserved
         if (newc>maxc) { t:=1 }             ;beyond the end of the data area
         if (t=0) {
            found:=1
            quit
         }
         j:=j+1
         if (j>=128) { quit }
      }
      if (found=1) {                         ;mark newc as end of chain (0FFFFFFF)
         half:=j/64                          ;idx points to byte 3 of the entry
         if (half=0) {
            t:=buf_lo[idx]
            t:=t&240                         ;keep the 4 reserved bits
            t:=t+15                          ;(low nibble is 0 now, + acts as OR)
            buf_lo[idx]:=t
            idx:=idx-1
            buf_lo[idx]:=255
            idx:=idx-1
            buf_lo[idx]:=255
            idx:=idx-1
            buf_lo[idx]:=255
         }
         default {
            t:=buf_hi[idx]
            t:=t&240
            t:=t+15
            buf_hi[idx]:=t
            idx:=idx-1
            buf_hi[idx]:=255
            idx:=idx-1
            buf_hi[idx]:=255
            idx:=idx-1
            buf_hi[idx]:=255
         }
         putc("N") putc("E") putc("W") putc("Space") putc("C") putc("Equal")
         put_hex32(newc)
         put_crlf()
         state:=71
      }
      default {
         fi:=fi+1
         pm:=raw32(fi)
         if (pm>=fsz) {                      ;no free cluster at all
            putc("F") putc("U") putc("L") putc("L")
            put_crlf()
            fail:=1
         }
      }
   }
   altif (state=71) {                        ;FAT 1 written
      if (nf=2) { state:=72 }
      altif (empty=1) { state:=76 }
      default { state:=73 }
   }
   altif (state=72) {                        ;FAT 2 written
      if (empty=1) { state:=76 }
      default { state:=73 }
   }
   altif (state=73) {                        ;FAT[c] := newc (link the old last cluster)
      q:=raw8(c)
      q:=q&127
      half:=q/64
      idx:=q&63
      idx:=idx*4
      pm:=newc
      c0:=raw8(pm)
      pm:=pm/256
```

```
        c1:=raw8(pm)
        pm:=pm/256
        c2:=raw8(pm)
        pm:=pm/256
        c3:=raw8(pm)
        c3:=c3&15
        if (half=0) {
           buf_lo[idx]:=c0
           idx:=idx+1
           buf_lo[idx]:=c1
           idx:=idx+1
           buf_lo[idx]:=c2
           idx:=idx+1
           t:=buf_lo[idx]
           t:=t&240
           t:=t+c3                          ;c3 <= 15, + acts as OR
           buf_lo[idx]:=t
        }
        default {
           buf_hi[idx]:=c0
           idx:=idx+1
           buf_hi[idx]:=c1
           idx:=idx+1
           buf_hi[idx]:=c2
           idx:=idx+1
           t:=buf_hi[idx]
           t:=t&240
           t:=t+c3
           buf_hi[idx]:=t
        }
        state:=74
     }
     altif (state=74) {                     ;FAT 1 written
        if (nf=2) { state:=75 }
        default { state:=76 }
     }
     altif (state=75) { state:=76 }         ;FAT 2 written
     altif (state=8) {                      ;put the next bytes into the sector
        loop {
           if (nleft=0) { quit }
           if (bo>=512) { quit }
           t:=txt[ci]
           half:=high(bo)
           idx:=low(bo)
           if (half=0) { buf_lo[idx]:=t }
           default { buf_hi[idx]:=t }
           ci:=ci+1
           if (ci>=LEN) { ci:=0 }
           bo:=bo+1
           coff:=coff+1
           fsize:=fsize+1
           nleft:=nleft-1
        }
        state:=9
     }
     altif (state=9) {                      ;data sector written
        if (nleft=0) { state:=10 }
        altif (coff>=cbytes) { state:=6 }
        default { state:=8 }
     }
     altif (state=10) {                     ;build the complete 32-byte entry in ent[]
        take j from [0,11) {                ;8.3 name without the dot
           t:=target[j]
           ent[j]:=t
        }
        ent[11]:=32                         ;attribute 20 = archive (normal file)
        ent[12]:=0                          ;reserved (lower-case flags)
        ent[13]:=0                          ;creation time, 10 ms units
        ent[14]:=0                          ;creation time 12h00m00s = 6000
        ent[15]:=96
        ent[16]:=80                         ;creation date 2026-10-16 = 5D50
        ent[17]:=93
        ent[18]:=80                         ;last access date
        ent[19]:=93
        ent[22]:=0                          ;write time
        ent[23]:=96
        ent[24]:=80                         ;write date
        ent[25]:=93
        pm:=firstnew                        ;first cluster, low word 26/27, high 20/21
        t:=raw8(pm)
        ent[26]:=t
        pm:=pm/256
        t:=raw8(pm)
        ent[27]:=t
        pm:=pm/256
        t:=raw8(pm)
        ent[20]:=t
```

```
        pm:=pm/256
        t:=raw8(pm)
        ent[21]:=t
        pm:=fsize                               ;size, offset 28..31
        t:=raw8(pm)
        ent[28]:=t
        pm:=pm/256
        t:=raw8(pm)
        ent[29]:=t
        pm:=pm/256
        t:=raw8(pm)
        ent[30]:=t
        pm:=pm/256
        t:=raw8(pm)
        ent[31]:=t
        half:=de/8                              ;copy ent[] into the free slot de
        base:=de%8
        base:=base*32
        take j from [0,32) {
           idx:=base+j
           t:=ent[j]
           if (half=0) { buf_lo[idx]:=t }
           default { buf_hi[idx]:=t }
        }
        state:=11
     }
     altif (state=11) {                         ;directory sector written
        putc("S") putc("I") putc("Z") putc("E") putc("Equal")
        put_hex32(fsize)
        putc("Space") putc("O") putc("K")
        put_crlf()
        state:=99
     }
     default { fail:=1 }                        ;unknown state

     ; state 20 (internal) = start a new, empty file in the free slot
     if (state=20) {
        dlba:=fdlba
        de:=fde
        c:=0
        fsize:=0
        empty:=1
        coff:=cbytes                            ;behaves like a full cluster -> allocate
        putc("F") putc("R") putc("E") putc("E") putc("Space") putc("L") putc("B") putc("A")
       putc("Equal")
        put_hex32(dlba)
        putc("Space") putc("E") putc("Equal")
        put_hex(de)
        put_crlf()
        state:=6
     }

     ; state 76 (internal) = the new cluster becomes the last one
     if (state=76) {
        if (empty=1) {
           firstnew:=newc
           empty:=2                             ;entry still needs the first cluster
        }
        c:=newc
        coff:=0
        state:=8
     }
     if (state=99) { quit }
  }

  if (fail=0) { setbit(LEDPORT,GREEN) }
  default { setbit(LEDPORT,RED) }
  loop { delay(1000) }                          ;done, wait for reset
```

A.10 SPI snippets in `_pic18Fxx_.jan`

```
;----------------------------SPI (MSSP master)------------------
; MSSP as SPI master, mode 0 (CPOL=0, CPHA=0), as SD cards need it
;   CKP=0 (clock idles low), CKE=1 (SDO changes on the active->idle edge, so the
;   data is stable on the rising edge, where the slave samples), SMP=0 (input
;   sampled in the middle of the data output time).
; Clock (SSPM3..SSPM0) = SPI_SLOW Fosc/64 (250 kHz at 16 MHz, SD init needs <=400 kHz),
;   SPI_MEDIUM Fosc/16 (1 MHz), SPI_FAST Fosc/4 (4 MHz).
; Pins = .chip_spi_pins() (18F452 RC3 = SCK out, RC4 = SDI in, RC5 = SDO out).
; The chip-select line is NOT handled here. It is an ordinary output pin in the
; user program (setbit/clearbit), because a later program may drive several slaves.
; BF (SSPSTAT bit 0) = a byte has been shifted in; reading SSPBUF clears it.
; NB names: no MyBug keyword inside snippet words (HOWTO-009 rule 8), hence "clock_bits",
; not "mode_bits" (contains mod).
```

```
14
15 <snippet> .spi_clock_bits(Speed)
16    <if> (Speed=SPI_SLOW)
17       #MOVLW 34              ;SSPEN=1, CKP=0, SSPM=0010 (Fosc/64)
18    </if>
19    <altif> (Speed=SPI_MEDIUM)
20       #MOVLW 33              ;SSPEN=1, CKP=0, SSPM=0001 (Fosc/16)
21    </altif>
22    <default>
23       #MOVLW 32              ;SSPEN=1, CKP=0, SSPM=0000 (Fosc/4)
24    </default>
25 </snippet>
26
27 <snippet> .configure_spi(Speed)
28    ; Speed = SPI_SLOW, SPI_MEDIUM or SPI_FAST
29    .chip_spi_pins()
30    #BCF SSPCON1,SSPEN,0      ;MSSP off while it is set up
31    #MOVLW 64                 ;SMP=0, CKE=1
32    #MOVWF SSPSTAT,0
33    .spi_clock_bits(Speed)
34    #MOVWF SSPCON1,0          ;also clears WCOL/SSPOV, switches the MSSP on
35 </snippet>
36
37 <snippet> .spi_set_speed(Speed)
38    ; changes only the clock (e.g. SPI_SLOW for the card init, then SPI_FAST)
39    #BCF SSPCON1,SSPEN,0
40    .spi_clock_bits(Speed)
41    #MOVWF SSPCON1,0
42 </snippet>
43
44 <snippet> .spi_xfer($Out,$In,$ref,$Local)
45 ; full duplex = sends the byte $Out and stores the byte received meanwhile in $In
46 ; ($Out and $In may be the same variable). Blocks for 8 SPI clocks.
47    #BANKSEL $Out_$ref
48    #MOVF $Out_$ref,W,1
49    #MOVWF SSPBUF,0           ;writing SSPBUF starts the 8 clocks
50    #spi_busy_$Local:
51    #BTFSS SSPSTAT,BF,0
52    #GOTO spi_busy_$Local
53    #MOVF SSPBUF,W,0          ;received byte, clears BF
54    #BANKSEL $In_$ref
55    #MOVWF $In_$ref,1
56 </snippet>
```

A.11 SPI pins in `_specific_pic18F452.jan`

```
1 ;---------------------------SPI pins (MSSP)-------------------------------
2 <snippet> .chip_spi_pins()
3 ; RC3 = SCK (output, master), RC4 = SDI (input), RC5 = SDO (output).
4 ; NB SDI is a Schmitt-trigger input, VIH min 0.7 VDD = 3.5 V (DS39564C DO41), so a 3.3 V
      slave
5 ; (SD card) needs an up-shifter on its MISO line when the PIC runs at 5 V.
6    #BCF TRISC,3,0
7    #BSF TRISC,4,0
8    #BCF TRISC,5,0
9 </snippet>
```