A screenshot of a computer

Description automatically generated

**The Pico In MicroPython: Direct To The Hardware**

Just occasionally you need to work directly with the hardware and MicroPython has the commands to let you do this fairly easily but you need to know more about the hardware than usual. This is an extract from our latest book all about the Raspberry Pi Pico in MicroPython.

MicroPython provides classes and methods to let you access most of the major hardware features of the Pico. They are very simple wrappers around the basic mechanism of working with the hardware – memory-mapped registers. Unfortunately at the time of writing there are many hardware features which are simply not exposed via MicroPython. In most cases it is possible to extend what you access using lower-level interactions with the hardware – still staying in MicroPython, but writing and reading the low-level register based hardware.

The obvious reason for knowing how to use memory-mapped registers is that if MicroPython doesn’t provide a function that does what you want, create it! Perhaps a better reason is just to know how things work. In this chapter we take a look at how the Pico presents its hardware for you to use and how to access it via basic software.

**Registers**

Some processors have special ways of connecting devices, but the Pico’s processor uses the more common memory-mapping approach. In this, each external device is represented by a set of memory locations or “registers” that control it. Each bit in the register controls some aspect of the way the device behaves. Groups of bits also can be interpreted as short integers which set operating values or modes.

How do you access a register? MicroPython provides a number of ways of doing this but the simplest is to make use of the mem functions in the machine module:

machine.mem32[address] Returns or sets a 32-bit value at the address

machine.mem16[address] Returns or sets a 16-bit value at the address

machine.mem8[address] Returns or sets an 8-bit value at the address

The only difficult part is in working out the address you need to use and the value that sets or resets the bits you need to modify.

For example, if you look in the documentation you will find that the GPIO registers start at address 0x40014000. The registers are defined by their offset from this starting address. So for example, the table of GPIO registers is:

|  |  |  |
| --- | --- | --- |
| **Offset** | **Register Name** | **Description** |
| 0x000 | GPIO0\_STATUS | GPIO status |
| 0x004 | GPIO0\_CTRL | GPIO control including function select and overrides |
| 0X008 | GPIO1\_STATUS | GPIO status |
| 0x00c | GPIO1\_CTRL | GPIO control including function select and overrides |
| … and so on down to | | |
| 0x0ec | GPIO29\_CTRL | GPIO control including function select and overrides |

You can see that there are two registers for each GPIO line from GP0 to GP29, one control register and one status register.

Each register has the same format for each GPIO line. For example, the status register is:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Bits** | **Name** | **Description** | **Type** | **Reset** |
| 31:27 | Reserved |  | - | - |
| 26 | IRQTOPROC | Interrupt to processors, after override applied | RO | 0x0 |
| 25 | Reserved |  | - | - |
| 24 | IRQFROMPAD | Interrupt from pad, before override applied | RO | 0x0 |
| 23:20 | Reserved |  | - | - |
| 19 | INTOPERI | Input signal to peripheral, after override applied | RO | 0x0 |
| 18 | Reserved |  | - | - |
| 17 | INFROMPAD | Input signal from pad, before override applied | RO | 0x0 |
| 16:14 | Reserved |  | - | - |
| 13 | OETOPAD | Output enable to pad, after override applied | RO | 0x0 |
| 12 | OEFROMPERI | Output enable from selected peripheral, before override applied | RO | 0x0 |
| 11:10 | Reserved |  | - | - |
| 9 | OUTTOPAD | Output signal to pad after override applied | RO | 0x0 |
| 8 | OUTFROMPERI | Output signal from selected peripheral, before override applied | RO | 0x0 |
| 7:0 | Reserved |  | - | - |

You can see that many of the 32 bits in the register are not used, but bit 9 is OUTTOPAD which is the final state of the GPIO line after register overrides have been applied. You can read its current value using:

from machine import mem32

addrGP0Status= 0x40014000

value=mem32[addrGP0Status]

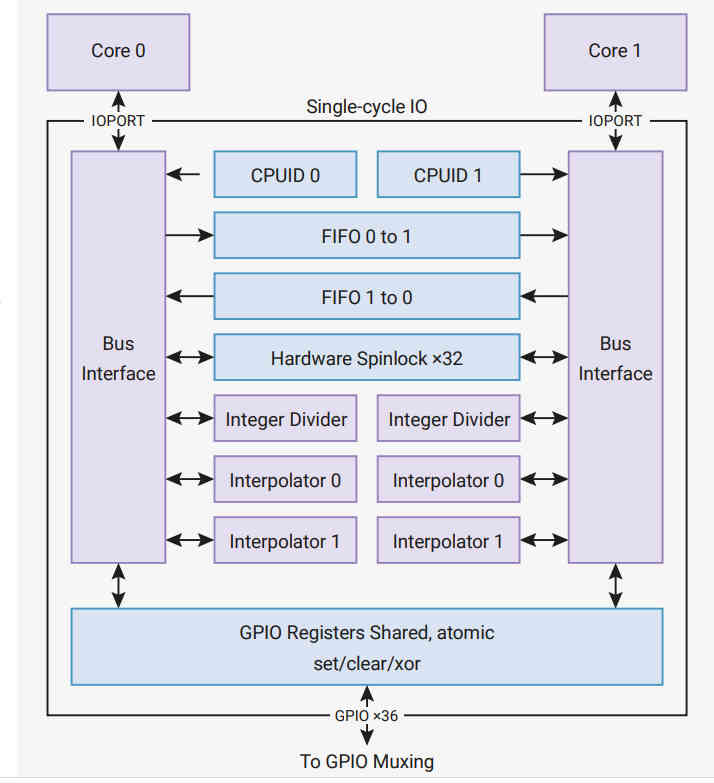
print(bin(value))

This prints the current status of GP0 in binary. If you want to find the status of GP*n* you need to use address 0x40014000+2*n*. Usually addresses are specified as a base address, i.e. where things start, and an offset that has to be added to the base to get the address of a specific device.

This is the general way you work with peripheral devices such as the PWM units or I2C hardware, but the GPIO is special in that it has another set of registers that control it.

## Single-Cycle IO Block

At this point you might think that we are ready to access the state of the GPIO lines for general input and output. This isn’t quite the whole story. To accommodate the fact that the processor has two cores, and to make access faster to important devices, there is a special connection, the SIO or Single-cycle IO Block, between the cores and, among other things, the GPIO. The SIO connects directly to the two cores and they can perform single-cycle 32‑bit reads and writes to any register in the SIO. Notice that the SIO is not connected via the general address bus. You can see the general structure of the SIO in the diagram below. You can find out about the other devices it connects to from the documentation - our focus is on the GPIO lines.



Notice that the GPIO lines are multipurpose and the SIO only has control when they are being used as GPIO lines. In this sense the SIO is just another peripheral that can take control of a GPIO line.

The SIO provides a set of registers that makes using the GPIO much faster and much easier. The basic registers are:

GPIO\_OUT Sets all GPIO lines to high or low

GPIO\_IN Reads all GPIO lines

GPIO\_OE Sets any GPIO line to output driver or high impedance

There are also three registers – SET, CLR and XOR - that make working with GPIO\_OUT and GPIO\_OE easier. Each of these can be thought of as a mask that sets, clears or XORs bits in the corresponding register.

For example, GPIO\_OUT\_SET can be used to set just those bits in GPIO\_OUT that correspond to the positions that are set high.

The locations of these registers are as offsets from 0xd0000000:

|  |  |  |
| --- | --- | --- |
| **Offset** | **Name** | **Description** |
| 0x004 | GPIO\_IN | GPIO Input value |
| 0x010 | GPIO\_OUT | GPIO output value |
| 0x014 | GPIO\_OUT\_SET | GPIO output value set |
| 0x018 | GPIO\_OUT\_CLR | GPIO output value clear |
| 0x01c | GPIO\_OUT\_XOR | GPIO output value XOR |
| 0x020 | GPIO\_OE | GPIO output enable |
| 0x024 | GPIO\_OE\_SET | GPIO output enable set |
| 0x028 | GPIO\_OE\_CLR | GPIO output enable clear |
| 0x02c | GPIO\_OE\_XOR | GPIO output enable XOR |

## Blinky Revisited

Now we can re-write Blinky yet again, but this time using direct access to the SIO GPIO registers.

from machine import mem32,Pin

from time import sleep\_ms

led=Pin(25,mode=Pin.OUT)

addrSIO = 0xd0000000

while True:

mem32[addrSIO + 0x014] = 1 << 25

sleep\_ms(500)

mem32[addrSIO + 0x018] = 1 << 25

sleep\_ms(500)

This program uses the standard MicroPython class to set the GPIO line to SIO control and output. If you think that this is cheating, it is an exercise to set the line correctly using the GPIO control register and the SIO.

This example is a demonstration rather than being useful, but there are some very useful functions we can write using our knowledge of how the GPIO lines are controlled. MicroPython is limited to controlling a single GPIO line at a time, but the hardware can change or read multiple GPIO lines in a single register operation. For example:

def gpio\_get():

return mem32[0xd0000000+0x010]

Here the get function simply reads the GPIO\_OUT register which has a single bit for the output state of each GPIO line. Notice that GPIO lines set to output reflect their last written-to state.

A set function simply writes the mask to the GPIO\_OUT\_SET register

def gpio\_set(mask):

mem32[0xd0000000+0x014] = mask

A clear function is just as easy and this just writes to the GPIO\_OUT\_CLR register:

def gpio\_clear(mask):

mem32[0xd0000000+0x18C] = mask

You can easily create functions for reset and other logical operations on all of the GPIO lines in one operation, but a single mask value function is usually sufficient:

def gpio\_set(value,mask):

mem32[0xd0000000+0x01C] =   
 (machine.mem32[0xd0000000+0x010])^value & mask

This writes to the GPIO\_OUT\_XOR register, but it writes a combination of a mask and a value. The mask gives the GPIO lines that need to be changed and the value gives the state they are to be set to. For example, if mask is 0111 and value is 0100 then value & mask is 0100. If this is XORed with the current state of the lines – e.g. 0101, in this case the result is 0001, which changes the state of only GP0 to a zero. Thus we have set lines GP2, GP1 and GP0 as specified in the mask to the corresponding bits in the value, i.e. 0100. Notice that this process sets the lines selected in the mask to either a zero or a one as determined by the bits in value.

As demonstrated in Chapter 4, the value,mask function can be used to set GPIO lines simultaneously:

from machine import Pin

import machine

def gpio\_get():

return machine.mem32[0xd0000000+0x010]

def gpio\_set(value,mask):

machine.mem32[0xd0000000+0x01C]=  
 (machine.mem32[0xd0000000+0x010])^value & mask

pin=Pin(22,Pin.OUT)

pin=Pin(21,Pin.OUT)

value1=1<<22 | 0<<21

value2=0<<22 | 1<<21

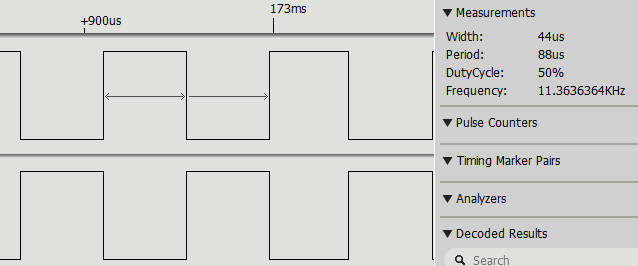
mask=1<<22 | 1<<21

while True:

gpio\_set(value1,mask)

gpio\_set(value2,mask)

This sets lines GP21 and GP22 to 01 and 10 on each pass through the loop:



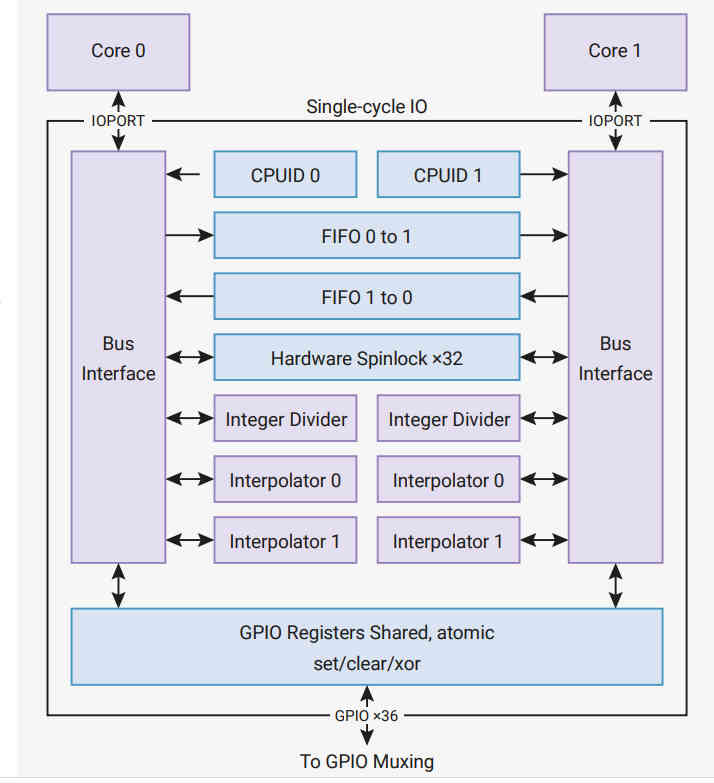
#### 

#### In Chapter But Not In This Extract

* Example I - Events
* Example II PAD - Pull, Drive and Schmitt
* Digging Deeper

## Summary

* All of the peripherals, including the GPIO lines, are controlled by registers – special memory locations that you write and read to configure and use the hardware.
* Exactly where the registers are positioned in address space is given in the documentation as a base address used for all of the similar registers and an offset that has to be added to the base to get the address of a particular register.
* The SIO block provides a more convenient way to access the GPIO lines and it has a different set of addresses and registers to the GPIO lines.
* With knowledge of how things work, you can add functions that are missing from MicroPython such as events and PAD control.
* Each GPIO line connects to the outside world via a PAD which has a number of configurable elements such as pull-up, slew rate and so on.



**Raspberry Pi Pico File System & SD Card Reader**

The Pico has 2MB of flash memory which is used to store the MicroPython interpreter. The rest of the memory is converted into a file system that you can make use of from within your own MicroPython programs.

The flash memory is divided into a number of partitions that hold the system and data. Unlike some other implementations of MicroPython, only the data partition,1.6MB, is available and it is automatically mounted as the root when the system boots up. You can gain access to this partition using the rp2.Flash() function which returns a Partition object representing the data partition.

Once you have an instance of Partition you can use any of the following methods:

part.readblocks(block\_num, buf, offset)

part.writeblocks(block\_num, buf, offset)

part.ioctl(cmd, arg)

These methods implement both the simple and extended the block protocols defined by os.AbstractBlockDev. Only a subset of ioctl commands are implemented for the Pico:

* 4 – Get a count of the number of blocks, should return an integer (arg is unused)
* 5 – Get the number of bytes in a block, should return an integer, or None in which case the default value of 512 is used (arg is unused)
* 6 – Erase a block, arg is the block number to erase

Generally these commands are too low level to work with a partition and instead you want to install a file system so that you can work in terms of files. But if you really want to reinvent the wheel then you can work directly in terms of raw blocks.

The Flash object is set up so that it isolates you from the other partitions and its block numbers start at zero. So to write some data to block zero and read it back we can use:

from rp2 import Flash  
import os

flash=Flash()  
os.umount("/")  
print(flash.ioctl(4,0))  
print(flash.ioctl(5,0))  
flash.ioctl(6,0)  
flash.writeblocks(0,b"Hello World")  
buf=bytearray(25)  
flash.readblocks(0,buf)  
print(buf)

First we unmount the file system so that it can't be used. If the partition was left mounted the file system could use the block we are about to use and overwrite it. Next we get the number of blocks and the block size. Before we can write new data to a block we have to erase it using ioctl command 6. After this we can write any number of bytes up to the block size. Reading the data back is just a matter of setting the length of the buffer to specify the number of bytes to read in. If you try this out you will find that we have stored "Hello World":

bytearray(b'Hello World\x00\x00\x00\x00\x008\  
 x01\x00\x00P\xa0\x00 \x0b')

The bytes beyond "Hello World" are whatever was already stored in the block.

**File Systems**

Working at the block level is fairly tedious and something you can generally avoid. As already mentioned, when MicroPython is first started it ensures that there is a file system installed in the partition and mounts it on the root ready for you to use.

The Pico supports two general file systems - a traditional FAT system and MicroPython’s own littlefs v2. You can create either type of file system on the data partition.The advantage of FAT is that it is a standard file system that can be read by other devices, but as we are using the internal Flash memory this isn’t relevant. FAT is more prone to errors than the alternative littlefs v2 which has the advantage of supporting wear leveling. For these reasons the Pico creates a littlefs v2 file system for you to use.

To work with a file system you first have to create it in a suitable partition – usually indicated by bdev:

os.VfsFat.mkfs(*block\_dev*)

creates a FAT file system and:

os.VfsLfs2.mkfs(*block\_dev*, readsize=32, progsize=32,   
 lookahead=32, mtime=True)

creates a littlefs v2 file system. Creating a file system on a partition is essentially formatting it and hence all existing data is lost.

Once you have created a file system it can be mounted either as the root file system or on any existing subdirectory:

os.mount(*fsobj*, *mount\_point*, \*, readonly)

To make modifications to the file system you have to unmount it:

os.umount(*mount\_point*)

You can install an alternative file system or just reformat the partition to remove all of the data. For example:

from rp2 import Flash  
import os

flash=Flash()  
os.umount('/')  
os.VfsLfs2.mkfs(flash)  
os.mount(flash, '/')

Most of the time you simply use a file system via the standard MicroPython os functions:

os.chdir(path) Change current directory  
os.getcwd() Get the current directory.  
os.ilistdir(dir) Iterate through directories returns  
 (name, type, inode[, size]):

os.listdir(dir) list the given directory.  
os.mkdir(path) Create a new directory.  
os.remove(path) Remove a file.  
os.rmdir(path) Remove a directory.  
os.rename(old\_path, new\_path) Rename a file.  
os.stat(path) Get the status of a file or directory.  
os.sync() Sync all filesystems.

There is also:

os.statvfs(path) Get the status of a filesystem.

which returns a tuple with the filesystem information in the following order:

f\_bsize – file system block size  
f\_frsize – fragment size  
f\_blocks – size of fs in f\_frsize units  
f\_bfree – number of free blocks  
f\_bavail – number of free blocks for unprivileged users  
f\_files – number of inodes  
f\_ffree – number of free inodes  
f\_favail – number of free inodes for unprivileged users  
f\_flag – mount flags  
f\_namemax – maximum filename length

Not all values are returned for a littlefs v2 file system.

You can also open a file and work with it using the standard stream functions:

read(), write(), readinto(), seek(), flush(), close()

For example:

from rp2 import Flash  
import os

print(os.listdir("/"))

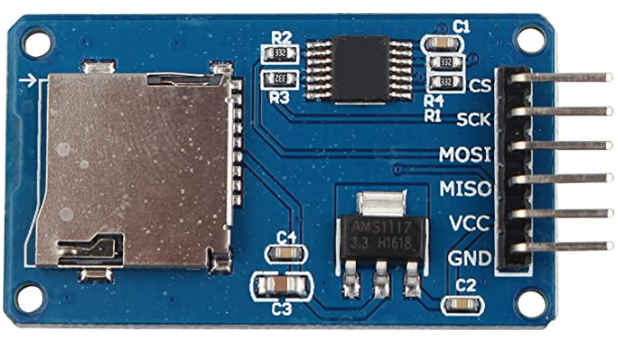
f=open("Hello.txt","wt")  
f.write("Hello World")  
f.close  
f.flush()  
print(os.listdir("/"))  
f=open("Hello.txt","rt")  
s=f.read()  
f.close  
print(s)

If you try this out you will discover that there is a new file called “Hello.txt” and you should see its contents displayed. Notice that you do need the flush as the system is buffered and if you simply open the file after closing it you will find that it is empty.

As this is non-volatile storage, you can use it to save state between boots.

**Adding an External SD Card Reader**

Although the Pico doesn’t have an SD card reader it is fairly easy to add one. Add-on SD card readers are available to order at very reasonable prices ($1.50):

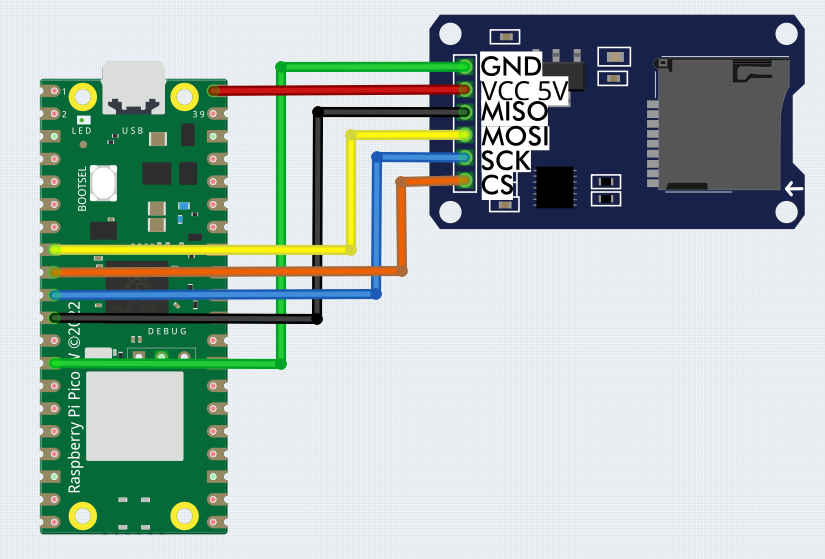


The only problem is that most have no documentation or specifications and they lack a card-detect and a write-protect pin. Connection to the Pico is fairly easy via one of the SPI buses. The only complication is that most of the devices need a 5V supply. They work at 3.3V logic levels and so can be directly connected to the Pico and have an onboard voltage regulator to reduce the supply to 3.3V. Most claim to work if powered from 3.3V but this depends on the regulator used and some fail or become unreliable,. The Pico has a suitable 5V supply pin in the form of VBUS but notice that this only works if it is being powered from the USB connection. In most cases this is the VCC connection to use.

Notice that you don't actually need an SD card reader as the pins on an SD card are nothing more than an SPI bus. The problem is you have to find a way to connect to them!

You can use any GPIO lines for the SPI connection and in this example the following are used:

|  |  |
| --- | --- |
| sck | GP10 |
| cs | GP9 |
| miso | GP11 |
| mosi | GP8 |



Once you have this wired up you need an SD card freshly formatted using FAT and a single partition – which is what you get if you use a new SD card. Make sure the card is correctly inserted before moving on to the software.

The SD Card driver isn't currently a default module in MicroPython. The simplest thing to do is to go to:

<https://github.com/micropython/micropython-lib/blob/master/micropython/drivers/storage/sdcard/sdcard.py>

and copy and paste the Python code into a file called sdcard.py. Upload this to the Pico and you should be able to include it in your program. To make use of it you first have to define the SPI interface:

spi = machine.SPI(1,  
                  baudrate=100000,  
                  polarity=0,  
                  phase=0,  
                  bits=8,  
                  firstbit=machine.SPI.MSB,  
                  sck=machine.Pin(10),  
                  mosi=machine.Pin(11),  
                  miso=machine.Pin(8))

and a GPIO line to use as CS:

cs = machine.Pin(9, machine.Pin.OUT)

You then use these to create an instance of sdCard:

sd = sdcard.SDCard(spi, cs)

The instance is a block device to which you can install a file system and then mount. For example, assuming that the SD card isn't formatted, you can format it to a FAT file system using:

os.VfsFat.mkfs(sd)

Notice that this will delete any data on the SD card and it takes few minutes to complete. Once you have a formatted card you can mount it:

os.mount(sd,"/sd")

The folder used as the mount point will be created if it doesn't exist. Now that the SD card is mounted we can read and write it using the standard file operations.

A complete test program that erases the SD card, writes some data and reads it back is:

import machine    
import os  
import sdcard  
cs = machine.Pin(9, machine.Pin.OUT)  
spi = machine.SPI(1,  
                  baudrate=100000,  
                  polarity=0,  
                  phase=0,  
                  bits=8,  
                  firstbit=machine.SPI.MSB,  
                  sck=machine.Pin(10),  
                  mosi=machine.Pin(11),  
                  miso=machine.Pin(8))  
sd = sdcard.SDCard(spi, cs)  
os.VfsFat.mkfs(sd)  
os.mount(sd,"/sd")

f=open("/sd/Hello.txt", "w")  
f.write("Hello World!\r\n")  
f.write("Some more data\r\n")  
f.close()

f=open("/sd/Hello.txt", "r")  
data = f.read()  
print(data)

If you don't want to format the card first comment out the line os.VfsFat.mkfs(sd).

If you find you have an SD card that doesn't work with this program try a lower clock speed. You can try increasing the clock speed to see if the SD card reader will cope. Some SD cards will fail to work at any speed as they don't implement the SPI bus in the correct way. As the SD card is FAT formatted it can be read in any machine that has an SD card slot.