Microcontroller Systems and Interfacing

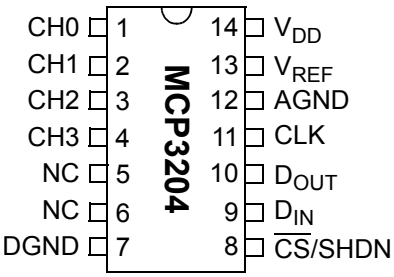
week 13 experimental lab

1 Serial Peripheral Interface (SPI) hardware

Complex devices (persistent memory and flash memory cards, D/A and A/D converters, real-time clocks, etc.) often have serial connections instead of parallel. They behave a little like a shift register, exchanging parallel information using one or two serial data signals and a clock. The two most common protocols are Inter-Integrated Circuit (I^2C) and Serial Peripheral Interface (SPI). The simpler (and in many ways the more flexible) of the two is SPI.

1.1 External ADC: MCP3204

To understand how SPI works, let's increase the resolution of our analogue input by connecting an external analogue-to-digital converter and communicating with it using SPI. A popular device for this is the MCP3204 which has four independent 12-bit ADC channels.



V_{DD} (pin 14) 5 V power supply

DGND (pin 7) 0 V digital ground
AGND (pin 12) 0 V analogue ground

V_{REF} (pin 13) reference voltage, sets the upper limit of input voltage (corresponding to the maximum digital A/D output value)

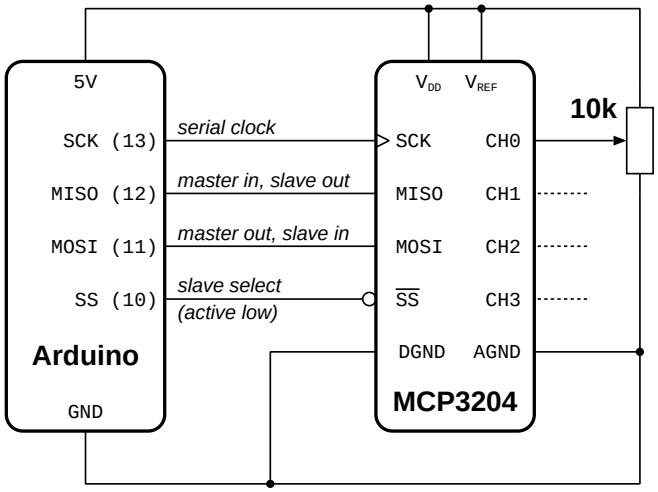
CH0–3 (pins 1–4) the four analogue input channels

CLK (pin 11) SPI serial clock input

D_{IN} (pin 9) SPI serial data input (equivalent to MOSI)

D_{OUT} (pin 10) SPI serial data output (equivalent to MISO)

$\overline{CS}$ (pin 8) SPI active-low chip select (equivalent to $\overline{SS}$)

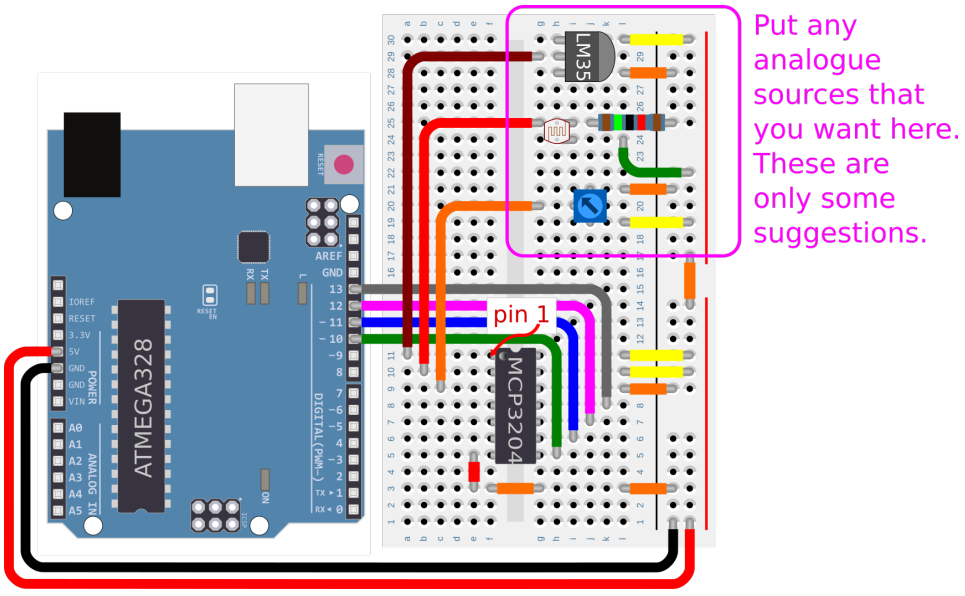


► Connect a MCP3204 ADC to the microcontroller. Use the following digital pins to carry the SPI signals:

SPI / Arduino		*MCP3204*
$\overline{SS}$	pin 10 → pin 8	$\overline{CS}$
MOSI	pin 11 → pin 9	D_{IN}
MISO	pin 12 → pin 10	D_{OUT}
SCK	pin 13 → pin 11	CLK

Be **very careful** to connect the MCP3204 the correct way around. Like the shift register, if the power is connected backwards the device **will** be damaged.

► Create a test voltage using a potentiometer (or a sensor of your choice) and connect it to the first input, CH0.



2 SPI communication software

Some microcontrollers (including the Arduino) provide SPI support in hardware. Access to the hardware's SPI signals is made available through a library. Other microcontrollers do not include harware SPI support, and communicating with SPI devices requires manual control of all four SPI signals. We will use both methods to communicate with the MCP3204.

2.1 Using the SPI library

To use a library, it is sufficient to include the associated header file at the beginning of the program. We will use the SPI library to quickly test our hardware.

- Create a new sketch. Make the SPI library available by including `SPI.h` at the beginning.

```
#include <SPI.h>
```

The SPI library manages the CLK, MISO and MOSI signals for us. It does not manage the slave select signal, so we will do that ourselves.

- Create a definition (using `#define` or `const int` to associate the symbol `SSN` ('slave select, active-low') with pin number 10.

```
const int SSN = 10; // slave select
```

- Create a `setup()` function that performs the following initialisation steps:

1. Start the `Serial` interface at 9600 baud.
2. Configure the `SSN` pin as an `OUTPUT`, and set its value to inactive.
3. Start the `SPI` interface using a clock divider of 4, SPI mode 0,0, and transfers with most significant bit first.

```
void setup() {
  Serial.begin(9600);
  pinMode(SSN, OUTPUT);
  digitalWrite(SSN, HIGH); // device inactive
  SPI.begin();
  SPI.setClockDivider(SPI_CLOCK_DIV16); // 1MHz
  SPI.setDataMode(SPI_MODE0); // low, rising
  SPI.setBitOrder(MSBFIRST);
}
```

- Create a function `int readADC(unsigned char channel)` that reads and returns a value from the given ADC `channel`. To read from the ADC using SPI, you will have to do the following steps:

1. Enable the ADC by setting `SSN` active.
2. Use `SPI.transfer()` to send (and receive) three bytes from the ADC. The first byte contains five zeros, a start bit 1, and mode bit 1 (single-ended conversion), and the most significant bit of the three-bit channel number. The second byte contains the least significant two bits of the channel number, followed by 6 zeros. The third byte can contain anything.
3. Collect the values returned by the second and third calls to `SPI.transfer()`. The least significant four bits of the first value are the four most significant bits of the ADC result. The second value contains the eight least significant bits of the result. Recombine them into a 12-bit result.
4. Turn off the ADC by setting `SSN` inactive.
5. Return the result from the ADC.

```
int readADC(unsigned char channel)
{
  digitalWrite(SSN, LOW);
  int adval = 0;
  SPI.transfer(0b00000110 + ((channel >> 2) & 1));
  adval =  ((unsigned char)SPI.transfer(channel << 6) & 0b00001111) << 8;
  adval |= ((unsigned char)SPI.transfer(0));
  digitalWrite(SSN, HIGH);
  return adval;
}
```

- Test your sketch with a `loop()` that prints the results of reading channel 0, then pauses for a quarter of a second. (Don't forget to open the serial monitor to view the output generated by the sketch.)

```
void loop()
{
  Serial.println(readADC(0));
  delay(250);
}
```

2.2 Implementing SPI manually

Let's simulate what would be needed on a microcontroller that does not have SPI hardware or a library giving access to it. This involves driving the SPI slave select, clock and data signals directly using `digitalWrite()`. (This technique is informally known as 'bit banging' the interface.)

- Save your sketch from the previous section under a new name. We are going to remove all references to the SPI library and generate the necessary SPI signals explicitly.
- Delete the `#include <SPI.h>` from the beginning of the sketch.
- Keep the defintion for `SSN`. Add three more definitions for `MOSI` (pin 11), `MISO` (pin 12), and `SCK` (pin 13).
- In `setup()`, delete the four lines that configure the `SPI` library. Replace them with code that performs the following actions:
  1. Configure MOSI as an `OUTPUT`.
  2. Configure MISO as an `INPUT`.
  3. Configure SCK as an `OUPUT` and set it to idle (`LOW`).
- Implement a function `void sendBit(int value)` that writes the least significant bits of `value` to the SPI device. You will have to perform two steps to do this:
  1. Write (using `digitalWrite()`) the *least significant* bit (only) of `value` to MOSI.
  2. Generate a positive pulse on SCK by setting it `HIGH` then back to `LOW`.
- Implement a function `int recvBit(void)` that reads one bit from the SPI device. You will have to perform four steps to do this:
  1. Set SCK active (`HIGH`) to ensure data is available on MISO.
  2. Use `digitalRead()` to obtain the value (0 or 1) of MISO.
  3. Set SCK inactive (`LOW`) to end the clock cycle.
  4. Return the bit that was read from MISO.
- Delete the three lines in `readADC()` that refer to `SPI.transfer()`. Replace them with code that explicitly initiates a conversion. (Refer to the timing diagram in Appendix B.) Your code will perform the following steps:
  1. Use `sendBit()` to send a `1` (start bit) followed by another `1` (single-ended mode) to the ADC. (We do not need to send the initial 5 zeros that were necessary when using the byte-oriented SPI library.)
  2. Use `sendBit()` to write the least significant three bits of `channel` to the ADC.
  3. Use `sendBit()` to write two 'don't care' bits to the ADC.
  4. Use `recvBit()` to read 12 bits from the ADC. For each bit read you will have to shift `adval` one bit left, then add to it the bit read from the ADC. At the end of this you will have reconstructed the 12 bits of ADC result.
  5. Return the final value of `adval`.

  Test your code. (You can leave the original `loop()` unmodified.)

3 Challenges

- Modify `loop()` (in either of your sketches) to display the values of all four ADC channels. Connect one or more voltage sources to the other ADC input channels and verify that the sketch prints the correct results.
- Disable (comment out) the `Serial.println()` and `delay()` from `loop()` in your two sketches. Modify loop to perform many conversions (call `readADC()` many times). Use `millis()` to record the time before and after performing the conversions. Calculate how many conversions per second are being peformed.

  SPI library conversions per second: _____ Manual SPI conversions per second: _____
- Connect a MCP4822 digital-to-analogue converter (DAC) and use it to generate a sine wave. (See the appendix for the relevant parts of the data sheet.) Note: V_{DD} must be connected to 5 V and V_{SS} must be connected to GND. Be **very** careful to get this right! To check the sine wave, connect the DAC output to an ADC input channel and display the results on the serial plotter. Note that you can share the SCK and MOSI signals between the ADC and the DAC but you will need two SPI protocols, two different 'slave select' pins (one for the ADC and a new one for the DAC), and a new $\overline{\text{LDAC}}$ signal for the DAC.

Microcontroller Systems and Interfacing

week 13 reference material

A Serial communication

A shift register can convert a single serial data signal (containing a fixed-length 'message' sequence of N single bits) and corresponding clock signal into N parallel signals. We can generalise this very simple protocol in several ways:

- connect the serial signal to more than one external converter device,
- send information in both directions (from microcontroller to device, or from device back to microcontroller),
- use variable-length 'message' sequences, etc.

Two popular general protocols are the Serial Peripheral Interface (SPI) and Inter-Integrated Circuit (I^2C). Both are 'in-system' protocols, meant for communication between components within a single piece of equipment.

Useful devices that use these protocols include Electrically-Erasable Programmable Read-Only Memory (EEPROM), Analog-to-Digital Converter (ADC), Digital-to-Analog Converter (DAC), real-time clocks (RTC), various sensors, Liquid Crystal Display (LCD) controllers, Secure Digital (SD) memory cards, the SM (system management) bus on Intel motherboards, etc.

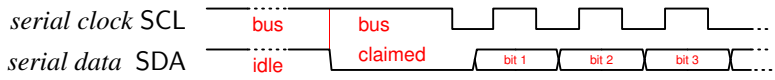
A.1 Inter-Integrated Circuit (I^2C)

I^2C was developed in 1982 by Philips Semiconductor to enable communication between digital devices in television sets.

I^2C is a *bus-based* protocol. Up to 127 devices can be connected to the bus, each having a unique 7-bit address. Devices communicate by exchanging byte-oriented messages with each other. I^2C therefore creates a 'mini network', in which senders and receivers are chosen dynamically according to a relatively complex protocol.

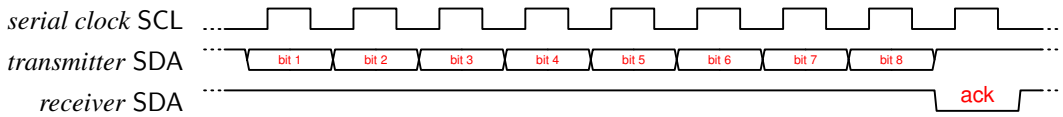
An I^2C bus needs just two wires: serial data SDA and serial clock SCL. The bus can be in one of two states, active or idle. When idle, both SDA and SCL are high, and no communication is taking place. When the bus is active, the clock cycles from high to low and back again while serial communication takes place on SDA. During a clock cycle, SDA is permitted to change only when SCL is low and must remain stable (for the receiver to sample) when SCL is high.

Any device can initiate a message send by (temporarily) claiming ownership of the I^2C bus while the bus is idle. Ownership is claimed by generating a falling edge on SDA while SCL is high, a condition that should never occur while the bus is active.

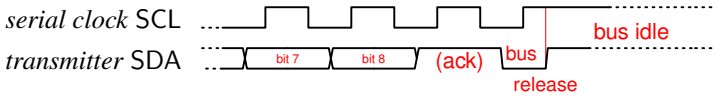


For the duration of the message exchange, the device that initiates the exchange is called the *master* and the device responding to the message is the *slave*. The master controls SCL throughout the message exchange. At various times during a message exchange, one of these devices is the transmitter and the other is the receiver. The transmitter controls SDA, regardless of which device is the master.

Every byte transmitted is followed by a single-bit acknowledgement from the destination device. Nine bits of data are therefore exchanged for each byte of data transmitted, with the receiver controlling the final acknowledgement bit.



At the end of data transfer, the master releases the bus by generating a rising edge on SDA while SCL is high, which (again) is a condition that never occurs while the bus is active.

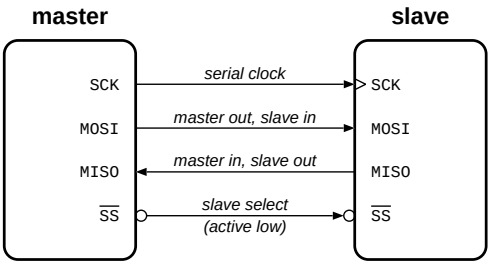


Each message begins with the master transmitting a byte containing seven address bits and one R/$\overline{\text{W}}$ 'direction' bit. The direction bit indicates if the master is writing to the slave or reading from it. The slave sends back a single bit acknowledging the receipt of the message. Master and slave then exchange an arbitrary number of data bytes, in the direction specified by the direction bit.

SDA ⌐			(bus claim)
seven-bit slave address	R/$\overline{\text{W}}$	ack	1× slave address and direction byte
eight data bits		ack	N× data bytes
SDA ⌐			(bus release)

I^{2}C permits two devices to communicate with each other in both directions, but only in one direction at a time (for a given message). It is therefore a *half-duplex* protocol. I^{2}C has relatively low performance, with typical maximum clock speeds of 100 kHz or 400 kHz. It is particularly good for configuring and monitoring devices that have control registers and/or status registers.

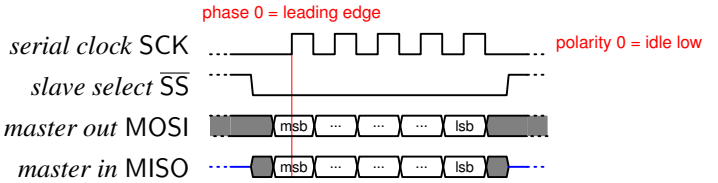
A.2 Serial Peripheral Interface (SPI)



SPI was developed by Motorola in 1985 for communication between a microcontroller (originally the M68HC11) and its external devices.

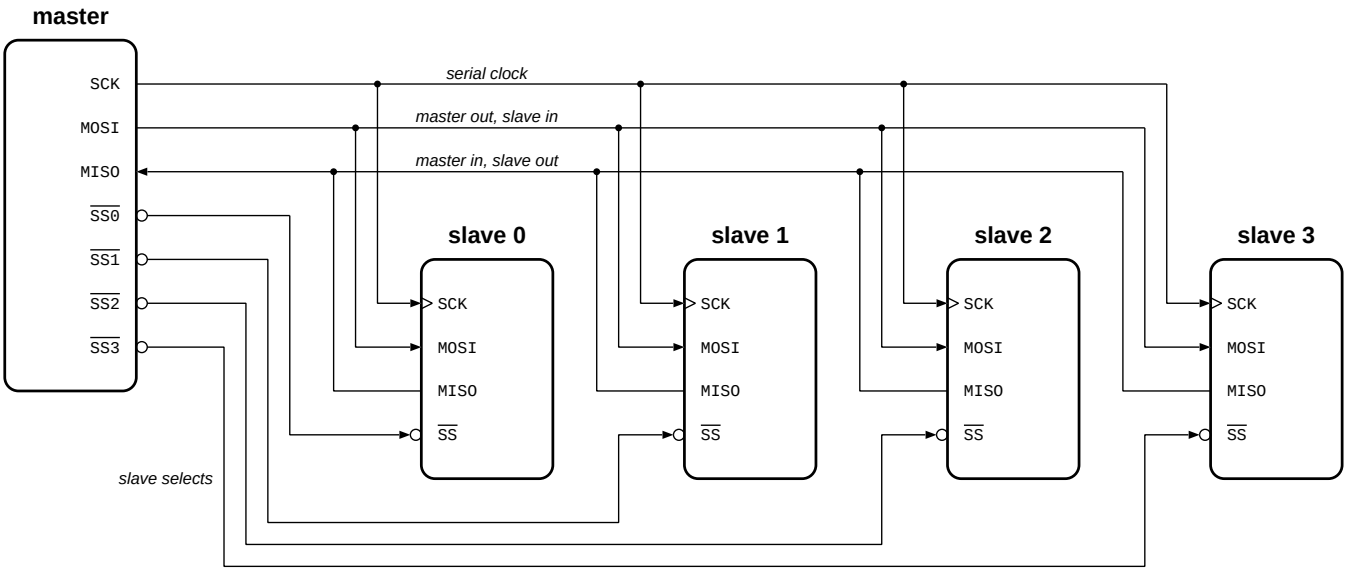
SPI is a *point-to-point* protocol. It connects a single master device to one or more slave devices. The slave devices have no address, but each one has an active-low 'slave select' ($\overline{\text{SS}}$) input. When $\overline{\text{SS}}$ is inactive (high), the slave device ignores its other SPI input signals and disables its SPI output (by making it high-impedance). This allows several slaves to share a single SPI connection, but only one of them can be active at a given time. The lack of addressing makes SPI a relatively simple protocol.

SPI needs four wires: an active-low slave select $\overline{\text{SS}}$, a serial clock SCK and two uni-directional data signals, master-out/slave-in MOSI and master-in/slave-out MISO. Several different conventions are possible for clocking, but the most usual is 'mode 0,0' in which the clock idles low (polarity 0) and received data signals are sampled on the leading (positive) edge of the clock signal (phase 0).

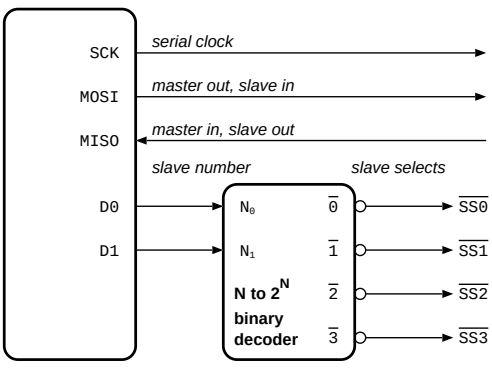


SPI clock mode (phase,polarity)	clock idles	active edge	
0,0	low	leading (rising)	⎍
0,1	low	trailing (falling)	⎍
1,0	high	leading (falling)	⎍
1,1	high	trailing (rising)	⎍

When using multiple slaves, each slave requires its own separate $\overline{\text{SS}}$ signal. SPI therefore requires $3 + N$ wires to communicate with N slave devices.



If N is large then the number of outputs required to generate the $\overline{\text{SS}}$ signals can be reduced using a serial to parallel converter (shift register) or a *binary decoder*. (A binary decoder has N inputs, representing a N-bit binary numeric value, and 2^N outputs of which only one is active at any time, selected according to the numeric value of the input number. It effectively converts a N-bit 'address' into N individual 'chip select' signals.)



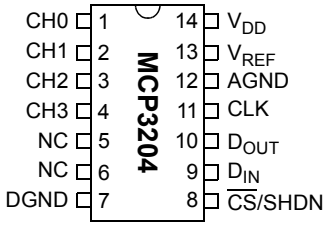
SPI permits the master and slave to exchange information in both directions at the same time. It is therefore a *full-duplex* protocol. SPI has relatively high performance, with typical maximum clock frequencies of 20–30 MHz (fast enough to transmit high-definition multi-channel audio, for example). It is particularly good for transferring streams of data, for example between a microcontroller and an external analogue-to-digital or digital-to-analogue converter.

B MCP3204: 4-channel 12-bit A/D converter with SPI

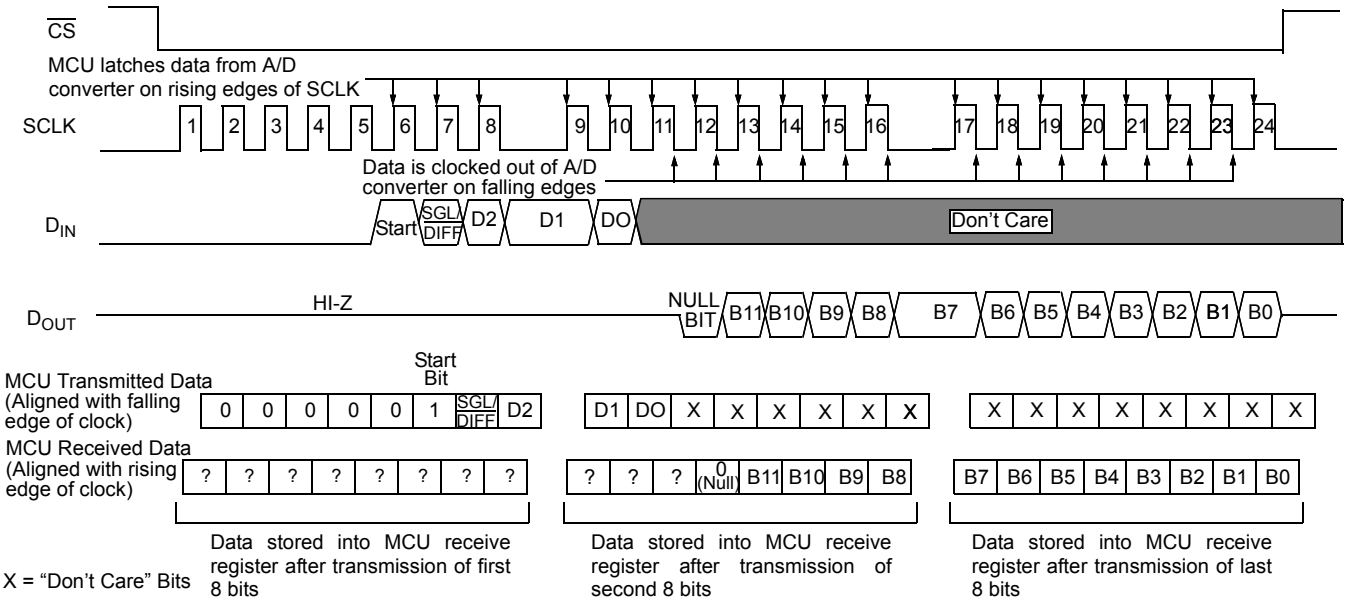
The 10-bit ADC on the Arduino can be too limited for some applications. Increasing the resolution to 12 bits is easy using an external ADC. (Higher resolution ADCs are available, but beyond 14 bits or so it becomes quite difficult to design and build circuits with low enough noise to make the extra bits useful.)

A popular device is the MCP3204, containing 4 separate 12-bit A/D converters. The maximum clock frequency is 2 MHz, or a maximum conversion rate of approximately 100,000 samples per second.

V_{DD}	5 V power supply
DGND	0 V digital ground
AGND	0 V analogue ground
V_{REF}	reference voltage, sets the upper limit of input voltage (corresponding to the maximum digital A/D output value)
CH0–CH4	the four analogue input channels
CLK	SPI serial clock input
D_{IN}	SPI serial data input (equivalent to MOSI)
D_{OUT}	SPI serial data output (equivalent to MISO)
$\overline{CS}$	SPI active-low chip select (equivalent to $\overline{SS}$)



To perform an A/D conversion, the number of the channel to be read is clocked into the device via MOSI as a five-bit value: a single ‘start bit’ (high), a single bit to select single-ended or differential mode, followed by three bits of binary channel number (for protocol compatibility with the MCP3208, which has eight input channels). The device then begins the requested conversion. Two clock cycles later it provides the result of the conversion on MISO as a 13-bit value, as one leading zero bit followed by the 12 result bits. The 12 bits of converted value are therefore sent back to the master device beginning three clock cycles after the address has been supplied, or seven cycles after the start bit was written.



When communicating over SPI one byte at a time (with the Arduino SPI library, for example) three bytes of data must be exchanged to initiate the conversion and then read back the result. From sending the start bit to receiving the last bit of converted result is a total of 19 clock cycles. Sending five zero bits before the start will therefore conveniently right-justify the result in the last 12 bits of the 24 that are read back.

MCP4802/4812/4822

8/10/12-Bit Dual Voltage Output Digital-to-Analog Converter with Internal V_{REF} and SPI Interface

Features

- MCP4802: Dual 8-Bit Voltage Output DAC
- MCP4812: Dual 10-Bit Voltage Output DAC
- MCP4822: Dual 12-Bit Voltage Output DAC
- Rail-to-Rail Output
- SPI Interface with 20 MHz Clock Support
- Simultaneous Latching of the Dual DACs with $\overline{LDAC}$ pin
- Fast Settling Time of 4.5 µs
- Selectable Unity or 2x Gain Output
- 2.048V Internal Voltage Reference
- 50 ppm/°C V_{REF} Temperature Coefficient
- 2.7V to 5.5V Single-Supply Operation
- Extended Temperature Range: -40°C to +125°C

Applications

- Set Point or Offset Trimming
- Sensor Calibration
- Precision Selectable Voltage Reference
- Portable Instrumentation (Battery-Powered)
- Calibration of Optical Communication Devices

Related Products(1)

P/N	DAC Resolution	No. of Channels	Voltage Reference (V_{REF})
MCP4801	8	1	Internal (2.048V)
MCP4811	10	1	
MCP4821	12	1	
MCP4802	**8**	**2**	
MCP4812	**10**	**2**	
MCP4822	**12**	**2**	
MCP4901	8	1	External
MCP4911	10	1	
MCP4921	12	1	
MCP4902	8	2	
MCP4912	10	2	
MCP4922	12	2	

Note 1: The products listed here have similar AC/DC performances.

Description

The MCP4802/4812/4822 devices are dual 8-bit, 10-bit and 12-bit buffered voltage output Digital-to-Analog Converters (DACs), respectively. The devices operate from a single 2.7V to 5.5V supply with SPI compatible Serial Peripheral Interface.

The devices have a high precision internal voltage reference (V_{REF} = 2.048V). The user can configure the full-scale range of the device to be 2.048V or 4.096V by setting the Gain Selection Option bit (gain of 1 of 2).

Each DAC channel can be operated in Active or Shutdown mode individually by setting the Configuration register bits. In Shutdown mode, most of the internal circuits in the shutdown channel are turned off for power savings and the output amplifier is configured to present a known high resistance output load (500 kΩ, typical).

The devices include double-buffered registers, allowing synchronous updates of two DAC outputs using the $\overline{LDAC}$ pin. These devices also incorporate a Power-on Reset (POR) circuit to ensure reliable power-up.

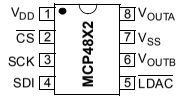
The devices utilize a resistive string architecture, with its inherent advantages of low DNL error, low ratio metric temperature coefficient and fast settling time. These devices are specified over the extended temperature range (+125°C).

The devices provide high accuracy and low noise performance for consumer and industrial applications where calibration or compensation of signals (such as temperature, pressure and humidity) are required.

The MCP4802/4812/4822 devices are available in the PDIP, SOIC and MSOP packages.

Package Types

8-Pin PDIP, SOIC, MSOP



MCP4802: 8-bit dual DAC
MCP4812: 10-bit dual DAC
MCP4822: 12-bit dual DAC

3.0 PIN DESCRIPTIONS

The descriptions of the pins are listed in Table 3-1.

TABLE 3-1: PIN FUNCTION TABLE FOR MCP4802/4812/4822

MCP4802/4812/4822 MSOP, PDIP, SOIC	Symbol	Description
1	V_{DD}	Supply Voltage Input (2.7V to 5.5V)
2	$\overline{CS}$	Chip Select Input
3	SCK	Serial Clock Input
4	SDI	Serial Data Input
5	$\overline{LDAC}$	Synchronization Input. This pin is used to transfer DAC settings (Input Registers) to the output registers (V_{OUT})
6	V_{OUTB}	DAC_B Output
7	V_{SS}	Ground reference point for all circuitry on the device
8	V_{OUTA}	DAC_A Output

3.1 Supply Voltage Pins (V_{DD}, V_{SS})

V_{DD} is the positive supply voltage input pin. The input supply voltage is relative to V_{SS} and can range from 2.7V to 5.5V. The power supply at the V_{DD} pin should be as clean as possible for a good DAC performance. It is recommended to use an appropriate bypass capacitor of about 0.1 µF (ceramic) to ground. An additional 10 µF capacitor (tantalum) in parallel is also recommended to further attenuate high-frequency noise present in application boards.

V_{SS} is the analog ground pin and the current return path of the device. The user must connect the V_{SS} pin to a ground plane through a low-impedance connection. If an analog ground path is available in the application Printed Circuit Board (PCB), it is highly recommended that the V_{SS} pin be tied to the analog ground path or isolated within an analog ground plane of the circuit board.

3.2 Chip Select ($\overline{CS}$)

$\overline{CS}$ is the Chip Select input pin, which requires an active-low to enable serial clock and data functions.

3.3 Serial Clock Input (SCK)

SCK is the SPI compatible serial clock input pin.

3.4 Serial Data Input (SDI)

SDI is the SPI compatible serial data input pin.

3.5 Latch DAC Input ($\overline{LDAC}$)

$\overline{LDAC}$ (latch DAC synchronization input) pin is used to transfer the input latch registers to their corresponding DAC registers (output latches, V_{OUT}). When this pin is low, both V_{OUTA} and V_{OUTB} are updated at the same time with their input register contents. This pin can be tied to low (V_{SS}) if the V_{OUT} update is desired at the rising edge of the $\overline{CS}$ pin. This pin can be driven by an external control device such as an MCU I/O pin.

3.6 Analog Outputs (V_{OUTA}, V_{OUTB})

V_{OUTA} is the DAC A output pin, and V_{OUTB} is the DAC B output pin. Each output has its own output amplifier. The full-scale range of the DAC output is from V_{SS} to G* V_{REF}, where G is the gain selection option (1x or 2x). The DAC analog output cannot go higher than the supply voltage (V_{DD}).

REGISTER 5-1: WRITE COMMAND REGISTER FOR MCP4822 (12-BIT DAC)

W-x	W-x	W-x	W-0	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
$\overline{A}$/B	—	$\overline{GA}$	$\overline{SHDN}$	D11	D10	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0
bit 15															bit 0

REGISTER 5-2: WRITE COMMAND REGISTER FOR MCP4812 (10-BIT DAC)

W-x	W-x	W-x	W-0	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
$\overline{A}$/B	—	$\overline{GA}$	$\overline{SHDN}$	D9	D8	D7	D6	D5	D4	D3	D2	D1	D0	x	x
bit 15															bit 0

REGISTER 5-3: WRITE COMMAND REGISTER FOR MCP4802 (8-BIT DAC)

W-x	W-x	W-x	W-0	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x	W-x
$\overline{A}$/B	—	$\overline{GA}$	$\overline{SHDN}$	D7	D6	D5	D4	D3	D2	D1	D0	x	x	x	x
bit 15															bit 0

Where:

bit 15 **$\overline{A}$/B:** DAC_A or DAC_B Selection bit
1 = Write to DAC_B
0 = Write to DAC_A

bit 14 — Don't Care

bit 13 **$\overline{GA}$:** Output Gain Selection bit
1 = 1x ($V_{OUT} = V_{REF} * D/4096$)
0 = 2x ($V_{OUT} = 2 * V_{REF} * D/4096$), where internal V_{REF} = 2.048V.

bit 12 **$\overline{SHDN}$:** Output Shutdown Control bit
1 = Active mode operation. V_{OUT} is available.
0 = Shutdown the selected DAC channel. Analog output is not available at the channel that was shut down. V_{OUT} pin is connected to 500 kΩ (typical).

bit 11-0 **D11:D0:** DAC Input Data bits. Bit x is ignored.

Legend			
R = Readable bit	W = Writable bit	U = Unimplemented bit, read as '0'	
-n = Value at POR	1 = bit is set	0 = bit is cleared	x = bit is unknown

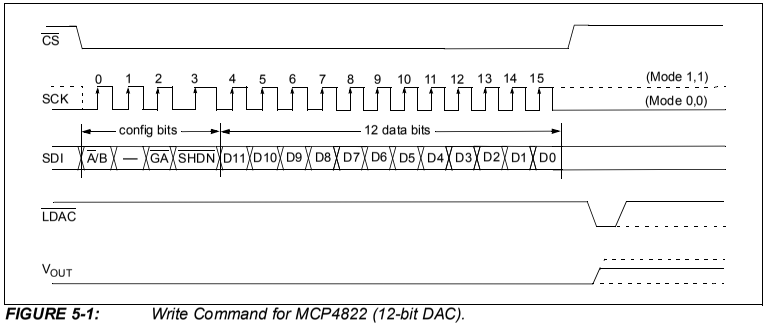


FIGURE 5-1: *Write Command for MCP4822 (12-bit DAC).*

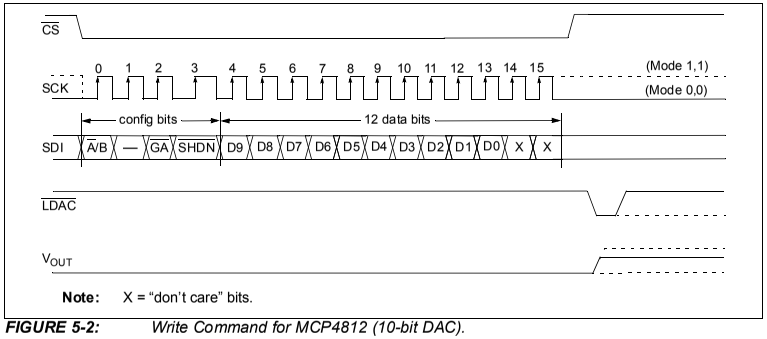


FIGURE 5-2: *Write Command for MCP4812 (10-bit DAC).*

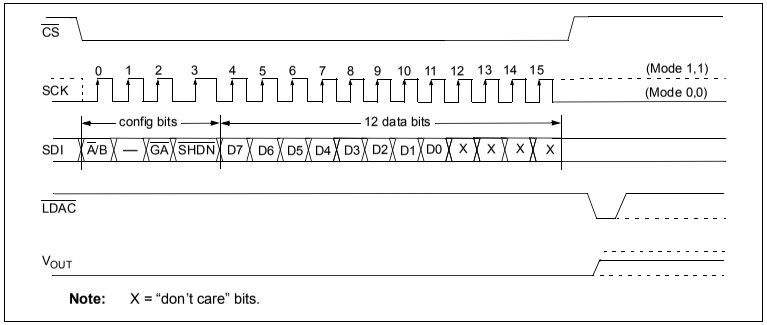


FIGURE 5-3: *Write Command for MCP4802 (8-bit DAC).*

Example solution to 'bit banging' exercise

```
#define SSN  10 // slave select pin
#define MOSI 11 // master out pin
#define MISO 12 // master in pin
#define SCK  13 // serial clock pin

void setup()
{
  Serial.begin(9600);
  pinMode(SSN, OUTPUT);
  digitalWrite(SSN, HIGH);      // slave select inactive
  pinMode(MOSI, OUTPUT);
  pinMode(MISO, INPUT);
  pinMode(SCK,  OUTPUT);
  digitalWrite(SCK, LOW);       // clock idle
}

void sendBit(unsigned char bit)
{
  digitalWrite(MOSI, bit & 1);  // value to write
  digitalWrite(SCK, HIGH);      // clock data into device
  digitalWrite(SCK, LOW);       // clock idle
}

int recvBit(void)
{
  digitalWrite(SCK, HIGH);      // clock don't care bit into device
  int bit = digitalRead(MISO);  // result bit from device
  digitalWrite(SCK, LOW);       // clock idle
  return bit;
}

int readADC(unsigned char channel)
{
  digitalWrite(SSN, LOW);       // slave select active

  sendBit(1);                   // start bit
  sendBit(1);                   // single-ended mode
  sendBit(channel >> 2);
  sendBit(channel >> 1);
  sendBit(channel);
  sendBit(0);                   // discard empty result bit
  sendBit(0);                   // discard null result bit

  int advalue = 0;
  for (int i= 0;  i < 12;  ++i)
    advalue = (advalue << 1) + recvBit();

  digitalWrite(SSN, HIGH);      // slave select inactive

  return advalue;
}

void loop()
{
  Serial.println(readADC(0));
  delay(250);
}
```