Lecture 1

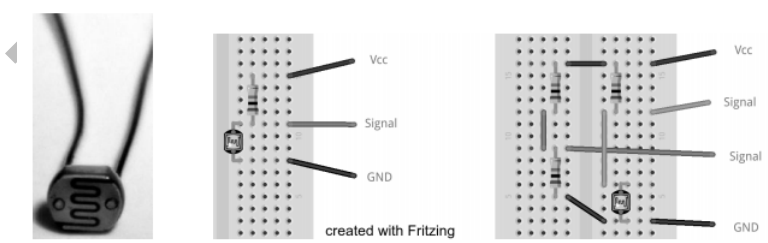
Sensors and Actuators-I

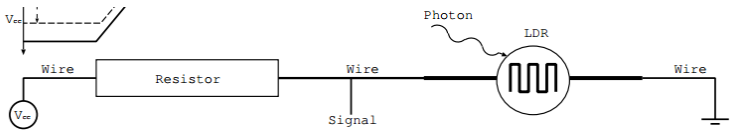
A sensor is a device that converts a physical quantity to an electrical signal [10, 11, 12, 13], and therefore either provides a voltage or a current, or causes a change of its resistance. More generally, the impedance of the sensor, which also comprises capacitive and inductive sensors, may change. We are thus faced with the task to measure either of these electrical quantities.

Analog Sensors

Resistance-based sensors

The first resistance-based sensor we look at is a light-dependent resistor or LDR, shown on the left of Figure 2.1. It changes its resistance depending on the exposure to light. The range of variation depends on the device and typically ranges from a few 100 ? to M?:





operating principle of this and many other sensors is based on the availability of electrons in the conduction band of a material. In good (wires) or bad (resistors)

conductors, electrons partially fill the available states in the conduction band up to some energy, the Fermi level, whereas in insulators the Fermi level is located in between the completely filled valence band and the conduction band. Therefore, no electrons are available in the conduction band.

Furthermore, the energy-difference between the upper boundary of the valence band and the lower boundary of the conduction band, the bandgap, is large, while for semiconductors it is on the order of electron-volt (eV).

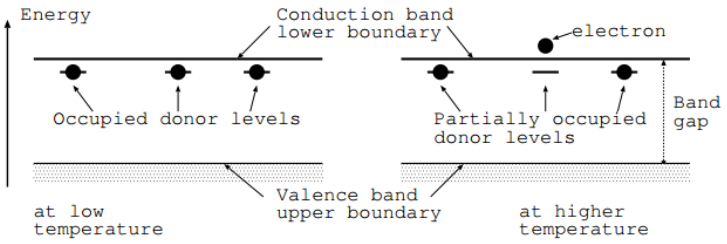
In photoresistors the base material is often CdS, a semiconductor with a bandgap of about 2.4 eV. This energy equals that of photons of green light with a wavelength of about 500 nm.

Other resistance-based sensors are **resistance-based temperature detectors (RTD)** such as the PT100 temperature sensor. It is a calibrated platinum-based sensor with a resistance of exactly 100 ? at 0 oC. It is based on the fact that the resistance of a very pure metal is determined only by scattering of electrons in the conduction band with phonons, which are vibrations of the ions that make up the crystal lattice of the metal. Moreover, higher temperatures cause stronger vibrations of the lattice, with correspondingly higher resistance.

Intuitively one might think of the crystal ions at higher temperature to oscillate with larger amplitudes, creating a larger target for the electrons to scatter, thus impeding their motion. Since this is an intrinsic property of the material, calibration measurements of resistance as a function of temperature are universally valid for all sensors of the same metal, provided the metal is very pure and free of imperfections.

Thermistors are resistors that have their temperature dependence deliberately made large. In positive temperature calibration (PTC) devices, the resistance increases with temperature, and in negative temperature calibration (NTC) devices, it decreases. PTCs are mostly used as protection devices that switch the resistance from a low- to a high-resistance state if a certain temperature is exceeded. They are

based on polycrystalline materials that change their dielectric constant at a certain temperature, the Curie temperature, by a large amount. Above the Curie temperature, the state of the magnetic dipoles is disordered and the dielectric constant is small. This causes the formation of large potential barriers between the crystal grains, which leads to a high resistance. Below the Curie temperature the molecular dipoles are aligned, the dielectric constant is large, and the resistance is low. A typical application of the PTC thermistor is a self-regulating heater, in which the heater also warms up the thermistor, which increases the resistance and limits the current to the heater until an equilibrium is found. PTCs can also be used to detect whether a threshold temperature is exceeded.



The converse thermistors are NTCs, which decrease their resistance with increasing temperature. They are often used for temperature sensing and are based on a doped semiconducting material that has occupied impurity donor levels below the conduction band, as shown in Figure 2.3. Increasing the temperature thermally excites these electrons to jump into the conduction band, thus increasing the conductivity. This effect is much larger than the reduction of the resistance due to the ions oscillating and impeding the motion of the electrons, which was responsible for the temperature dependence in the PT100 sensor. Both NTC and PTC thermistors are sensed by connecting them to a constant-current source and measuring the voltage drop across the thermistor

A number of position sensors are based on **potentiometers**. A potentiometer is a variable resistor where a slider moves up and down a resistance and shortens

the distance of one end point to the slider, thereby reducing the resistance between two terminals. The distance between the slider and the other end point lengthens, causing the resistance between the slider and the other terminal to increase correspondingly. On the left-hand side of Figure 2.4 we show a potentiometer with three connectors; the two end points are connected to dark wires and the one controlled by the slider is connected to a lighter-colored wire. The schematic view on the right of Figure 2.4 explains the functionality; the slider controls a variable midpoint of a voltage divider and the output voltage interpolates from 0 to 5 V in this case.

Voltage-based sensors

An example of a sensor that directly produces a voltage at its output pin is the LM35 temperature sensor, which is a silicon-bandgap temperature sensor. The operating principle is based on passing known currents In with n = 1; 2 with current densities jn across the base-emitter junctions of two bipolar transistors, and comparing their respective voltage drops VBE;n: The voltage difference is proportional to the temperature. This is easily understandable by inverting the currentvoltage curve for the diode of the base-emitter junction

$$j_n = A(T)\left[e^{(eV_{BE,n}-E_g)/\kappa T} - 1\right]$$

where $E_g = 1.2$ V is the bandgap energy of silicon, k is the Boltzmann constant, and T the absolute temperature in Kelvin. A(T) is a device-specific constant with moderate temperature dependence. Assuming that both transistors are located on

the same substrate and have the same temperature, we solve for two current densities j1 and j2 and obtain for the voltage difference

In the LM35, the baseemitter diodes of the two transistors have different areas such that the ratio of the areas determines the current densities, provided that the same macroscopic current passes through the two transistors. There are operational amplifiers on the same substrate to provide signal conditioning such that the LM35 produces an output voltage Vs that is related to the temperature T by Vs = T=100: Here Vs is measured in volts and the temperature in degrees Celsius, such that a temperature of 23 oC results in a voltage of 0.23 V. The LM35 has three pins; one is connected to ground, one to the supply voltage, and the third one carries the voltage Vs that is proportional to the temperature. Note the polarity for connecting the LM35 in Figure.

Thermocouples are temperature sensors that are based on the effects of temperature and temperature gradient on conductors made of different materials. Directly at the junction of the conductors, the Peltier effect causes a current that depends on the temperature. This happens at the points labeled by their respective temperatures T1 and T2 on the top left in Figure 2.9. On the wire segments a temperature gradient causes an additional current to flow, the Thomson effect. And finally, joining the two junctions and the wires causes a current to circulate, provided the loop is closed. This is called the Seebeck effect. If the loop is open, as shown at the top left of Figure 2.9, a voltage U develops at the end terminals as a consequence of the Peltier, Thomson, and Seebeck effects. In practice, one junction, say at T1, is held at known and constant temperature, for example, by immersing the

junction in ice water. Then the voltage U is related to the temperature difference T2-T1 of the sensing end at T2 and the reference temperature T1: The magnitude of the voltage generated depends on the combination of metals and is typically on

the order of 50 $\mu V = oC$. In a **thermopile** a number of wire segments of materials A and B are connected in series, as shown on the bottom left in Figure 2.9. This increases the sensitivity of the device to temperature differences. Thermopiles are often found in devices sensing heat and infrared radiation, such as thermal imaging devices or contact-free thermometers. The sensor used in the latter is shown on the right of Figure 2.9

Some crystals and ceramics react to external stresses by producing a **piezoelectric** voltage between opposite sides of the material, as a consequence of rearranging charges within their crystal structure. The resulting voltages reach several kV and can be used to produce sparks in ignition circuits or in old-fashioned vinyl record players. There, a "crystal" - stylus is squeezed in the grooves of the record and the generated voltages are amplified and made audible as sound. In scientific applications, piezoelectric sensors are used to measure pressure or forces.

Hall sensors produce a voltage that is proportional to the magnetic induction B: Their mode of operation is explained in Figure 2.10 and is based on passing a known current I through a semiconductor. In the presence of a magnetic field, the Lorentz force deflects the charge carriers—electrons and holes—towards perpendicularly mounted electrodes . This creates a potential difference (a voltage) between the electrodes, which causes a transverse electric field that counteracts the deflection from the Lorentz force such that the following charge carriers can move towards the exit electrode undeflected. In equilibrium, the voltage difference between the upper and lower electrodes is proportional to the magnetic induction B and can be measured with a voltmeter. The A1324 Hall sensor, shown on the right in Figure 2.10, has signal-conditioning circuitry on board and only needs three pins for ground, supply voltage, and output voltage. The latter is proportional to the magnetic field, with a sensitivity of 50 mV/mT centered at 2.5 V when no field is present

Lecture 2

Sensors and Actuator - II

Digital Sensors

We now address sensors that do not require an external ADC, but report their measurement values directly to the microcontroller in digital from. Some sensors already have the ADC built in, and others do not need one. We start with the latter, of which the most prominent examples are buttons and switches.

Buttons and switches

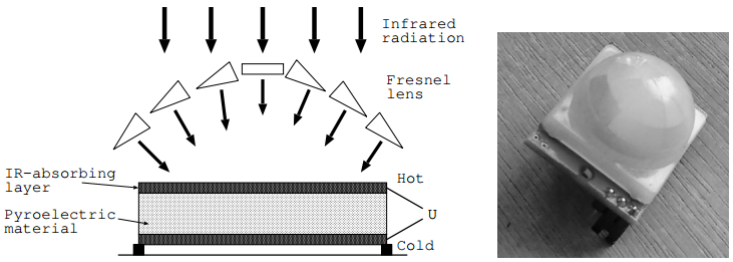
The simplest digital sensor is certainly a switch that is either closed or open, or a button where the open or close state is only activated temporarily. We use these terms interchangeably. Our task is to sense their state in a reliable way, and this is normally done with a pull-up resistor that is connected to the supply voltage in the way shown in Figure 2.31. In this way the sensing pin on the microcontroller can reliably detect the supply voltage. Only if the switch S1 is pressed does the voltage on the pin drop to zero or ground level. Only a small current, determined by the magnitude of the resistor, flows when the switch is closed. The actual value of the resistor is uncritical, but values around 10 to 30 k are usually reasonable. Without the pull-up resistor, the voltage potential on the pin is undefined when the switch is open, and determined by stray capacitances in the system. So, the recommendation is to always use a pull-up resistor when sensing the state of a switch. Note that swapping the position of the resistor and the switch, the resistor functions as a pull-down resistor, and the sensing level is zero unless the switch is closed.

On/off devices

A number of sensors provide a voltage level to inform the microcontroller about their state or change of state. They can be thought of as a switch with a built-in pull-up resistor and can be sensed in the same way. A problem can occur if the operating voltage level of the sensor and the microcontroller do not agree. Nowadays many sensors operate on levels of 2.5 to 3.3 V, and microcontrollers on levels from 2.5 to 5 V. Sensing higher external voltages such as those used in cars (12 V) or industrial control applications (24 or 48 V) requires some adjustment of the voltage level, to prevent damaging either the sensor or the microcontroller. There are level-changing chips available, such as the 74LVC245, but in many cases a simple voltage divider with two resistors is adequate. Yet, it only works if there is signal flowing from the high to the low-voltage side. In case a bidirectional signal flow is necessary, such as on the data line of the I2C bus, the solution shown in Figure 2.32, based on an n-type MOSFET transistor, is easy to implement. In the first case, when both logic signals are high, the MOSFET is not conducting, because the voltage difference between gate and source is close to zero. In the second case, if the 3.3 V logic is controlling and the signal is pulled low, the difference between gate and source is positive and the MOSFET conducts, such that even the 5 V logic level is pulled low. In the third case, when the 5 V logic is controlling and the 5 V logic signal is pulled low, the built-in diode (visible in the schematics) conducts, and causes the voltage of the source to drop to about 0.7 V. At this point the gate source voltage drop is sufficiently large to fully cause the MOSFET to conduct, which also pulls the 3.3 V logic level low.

But let us return to the sensors. A prominent device that reports its state through a changing voltage level is a PIR proximity sensor, shown on the right of Figure 2.33. It senses the change in the infrared radiation level, which announces the presence of living beings. The sensors are based on collecting the incident infrared radiation

with a Fresnel lens, which is the dome visible in Figure 2.33, on a pyroelectric sensor. The lens is often made of polyethylene, a material chosen for its low absorption of IR radiation. One side of the sensor is consequently warmed up and expands, which causes buckling of the piezo- or pyroelectric material, often a polymer film. Two effects contribute to the voltage between the upper and lower plates. First, the buckling strains the material and causes a piezoelectric voltage. Second, heat flows from the hot to the cold side and adds a pyroelectric voltage.



I2C devices

A large number of sensors have some logic built in and support a high-level communication protocol. An example is the I2C protocol operating on the I2C bus. The physical connection to devices supporting I2C only needs four wires: Ground, supply voltage Vcc, clock SCL, and data SDA. The latter two require a pull-up resistor, which is often already included in the microcontroller that also serves as the I2C busmaster to orchestrate the communication. It configures the sensor, initiates a measurement, and then reads data from the sensor. Physically, the communication is based on a synchronous serial protocol, where the data line is sampled every time the clock line changes from a high level to a low level. The protocol is standardized and we will not go into details, but mention that the I2C devices and also I2C sensors have a number of registers internal that can be written to in order to configure the sensor, or read from in order to retrieve sensor data. The communication is

entirely based on exchanging digital signals, and, as mentioned before, coordinated by the busmaster which normally is a microcontroller. Several devices can share the same SDA and SCL lines, because each device has its own address and responds only to those messages intended for it by specifying the device address.

SPI devices

The SPI interface is a synchronous serial communication bus, similar to the I2C bus, but it can operate at much higher speed and is therefore often used for devices that require the continuous transfer of large amounts of data, such as displays or audio equipment. SPI communication requires one master on the bus, a role normally taken by a microcontroller. The sensors are typically slave devices. They need at least six wires to connect: ground and supply voltage, the clock CLK, one line to send information from the master to the slave (MOSI, for master-out slave-in), one line for the reverse direction (MISO for master-in slave-out), and a chip-select line CS to identify the currently active slave. CLK, MISO, and MOSI lines can be shared among many slaves, but each slave requires its individual CS line.

RS-232 devices

Several devices report their measurement values by sending them via the asynchronous RS- 232 protocol. Originally, the physical medium for the communication channel used a current loop, but nowadays most sensors operate on 3.3 or 5 V levels. The communication happens point-to-point between two partners who have agreed on a communication speed, which is often 9600 baud or 115200 baud. Three wires are required at the minimum, one for ground potential, one, labeled TX for transmitting from device A to device B, and another one, labeled RX, for transmitting in the reverse direction. To establish communication, cables connect the TX pin on one device to RX on the other and vice-versa. Naturally, ground pins need to be

connected as well.

One device that supports RS-232 communication is the LV-EZx distance sensor we already discussed earlier. It can be configured to send the distance measured as an ASCII string that can be read in a terminal program.

Sensors that query the global positioning system (GPS) use a small patch antenna on the sensor to pick up signals from a number of satellites placed in geostationary orbits, which broadcast their position and timing information with high precision. On-board electronics that normally comprise a microcontroller use triangulation in order to determine the position of the sensor with high accuracy and convert that information to an ASCII string containing the position in a standardized format, called NMEA. The string is written, typically once per second, to an RS-232 serial line, where it is straightforward to read and decode.

Lecture 3

Arduino I

The original Arduinos are based on Atmel microcontrollers and we mostly discuss the Arduino UNO. Support for a second family of controllers, based on the ESP8266 microcontroller, was recently integrated into the Arduino development environment. These controllers can be programmed in much the same way as UNOs, but have native wireless support built in. But let?s start with the UNOs.

Arduino UNO

An Arduino UNO is shown in Figure 4.1, where the main component is the ATmega328p microcontroller from Atmel (now MicrochipTM), which is the large chip with 28 legs in the image. It is a controller with 8-bit-wide registers, and operates at a clock frequency of 16 MHz. It has 32 kB RAM memory and 1 kB non-volatile EEPROM memory, which can be used to store persistent data that need to survive tuning off and on the supply voltage. There are three timers on board, which are basically counters that count clock cycles and are programmable to perform some action, once a counter reaches some value. The UNO interacts with its environment through 13 digital inputoutput (IO) pins, of which most can be configured to be either input or output, and have software-configurable pull-up resistors. Several of the pins are configurable to support I2C, SPI, and RS-232 communication. Moreover, there are six analog input pins. They measure voltages of up to the supply voltage of 5 V. An alternative internal reference voltage source provides a 1.1 V reference. All digital and analog pins are routed to pin headers that are visible on both sides of the Arduino printed circuit board (PCB) in Figure 4.1. Furthermore, the built-in hardware RS-232 port is connected to an RS-232-to-USB converter that

allows communication and programming from a host computer. There is no WiFi, Bluetooth, or Ethernet support on the UNO board, but extension boards, so-called shields, are available for mounting directly onto the pin headers.

One can describe the Arduino UNO as having the intelligence of a washing machine. It keeps time with the timers, it can sense voltages from, for example, temperature sensors, and it can turn motors or pumps on or off, depending on whether some condition is met. In this way it can also provide the glue logic to interface sensors (analog, I2C, SPI, other) to the host computer, and that is the mode in which we will use the UNO later on.

Arduino IDE

Once the installation completes, we start the IDE by clicking the icon, or start it from the command line by typing arduino followed by Enter, which should open the IDE and show us a “indow similar to the one shown in Figure 4.3. If not, create a ”New” sketch, which is what Arduino programs are called, by selecting “New” from the “File” menu.

Here we already see the general structure of Arduino programs (or synonymously “sketches”). There is a setup() function, which is executed once, immediately after power is turned on. In this routine all initialization housekeeping is done, such as defining a pin to be output or input, and configuring the serial line. Once the setup() function completes, the loop() function is called repetitively, such that once it completes, it is called again and so forth, until power is turned off. Note that the programming language supported by the Arduino IDE is very similar to the C language. There are, however, a number of special extensions to provide access to the specific hardware, such as the ADC

Now we connect the Arduino UNO to any USB port on the host computer where the Arduino IDE is running, and select Arduino UNO from the Tools!Board menu.

This step tells the IDE for which processor the compiler will generate code, as well as some hardwarespecific definitions such as the names of IO pins that we can use when programming. At this point the IDE "knows" what the hardware is, but we still need to tell the IDE to which USB port the UNO is connected. This we do in the Tools!Port menu, where normally the serial port to which the UNO is connected automaticlly appears and can be selected. On Linux this often is /dev/ttyUSB0 or /dev/ttyACM0. On a Windows computer it is COMx where x is some number.

A Sample Program

```
void setup() {
pinMode(LED_BUILTIN, OUTPUT);
}
void loop() {
digitalWrite(LED_BUILTIN, HIGH); // turn the LED on
delay(1000); // wait for a second
digitalWrite(LED_BUILTIN, LOW); // turn the LED off
delay(1000); // wait for a second
}
```


In the setup() function, we call the pinMode(Pin,What) function, which declares that the pin called LED BUILTIN (which is pin 13 on a UNO) will be used as output pin. That is all the initializing we do in the setup() function. In the loop() we call digitalWrite(Pin,State), which causes the controller to put 5 V onto the specified pin if the requested state is HIGH and 0 V if the requested state is LOW. The latter happens two lines later. In between the changes to the output pin, we tell the controller to wait for a specified number of milliseconds in the delay(time in ms) function. So, all the loop function does is turn on the LED on pin 13, wait for 1000 ms or 1 s, turn it off, wait again for 1 s, and then start all over again. Note that all

commands are terminated by a semicolon, as is customary in the C-language.

Lecture 4

Arduino II

Interfacing Sensors

Button

In Figure 4.5, we show how to connect a button between ground and pin 2 of the Arduino, such that pressing the button will cause pin 2 to read 0 V or LOW. The following listing shows a sketch that causes the LED on pin 13 to light up, if the button is pressed.

```
// button_press, V. Ziemann, 161013
void setup() {
pinMode(2,INPUT_PULLUP); // button, default=HIGH
pinMode(13,OUTPUT); // LED
}
void loop() {
if (digitalRead(2)==LOW) {
digitalWrite(13,HIGH);
} else {
digitalWrite(13,LOW);
}
delay(10);
}
```

The program follows the normal scheme of initializing the hardware in the setup() function. We first declare that pin 2 is an input pin, and we enable the internal pull-up resistor that causes a well-defined "high" state on the pin if no button is

pressed. Pin 13 with the LED is declared as output, so we can turn it on and off from within the sketch. In the loop() function, we check the state of the button by calling the digitalRead(Pin) function, and test whether pin 2 is LOW. If this is the case (note the double equal sign == in the comparison), the LED on pin 13 is turned on by calling digitalWrite(13,HIGH). If pin 2 is found not to be LOW, the LED on pin 13 is tuned off with digitalWrite(13,LOW);. Note the braces f and g and their use to define blocks of code in the if (..) f g else f g construction. After the if statement, a short delay() ensures that mechanical button bounces are ignored and that the processor has a little time for its internal affairs. This is not absolutely necessary, but good style.

Analog Input

We place them in a voltage divider and then read the voltage change on the central tap, as shown on the left in Figure 2.1. Others, such as the LM35 temperature sensor, directly produce a voltage and in this section we show how to interface these sensors to the Arduino. First, we want to measure a voltage from one of the analog input pins; here we use A0. The left of Figure 4.6 shows how to connect a potentiometer as a variable voltage divider to the Arduino. One end of the potentiometer is connected to ground, the other end to the supply voltage, and the wiper with the variable tap of the potentiometer is connected to pin A0. Turning the axis of the potentiometer causes the voltage on the center tap to vary between ground and the 5 V supply voltage.

```
// Analog and serial communication, V. Ziemann, 161130
int inp,val;
void setup() {
Serial.begin(9600); // baud rate
}
```

```
void loop() {
if (Serial.available()) {
inp=Serial.read(); // read character from serial
val=analogRead(0); // read analog
Serial.print("Value is "); // and report back
Serial.println(val);
}
delay(50); // wait 50 ms
}
```

In the setup() function, we declare that we want to use the built-in serial (RS-232) hardware port with a speed of 9600 baud, which is about 1000 characters per second. The receive (RX) and transmit (TX) lines of the serial port are connected to pin 0 and pin 1, but also routed to the RS232-USB converter on the UNO board that is connected to the host computer. In the loop() function, we first check whether some communication from the host computer has arrived, and if that is the case, we read the character with the Serial.read(); command, but do nothing further with it. In the next line we read the analog pin A0 using the built-in ADC, and store the value in the integer variable val. The value returned from the 10-bit ADC is a number between 0 and 1023 (= 210 ? 1) and not the voltage. Later we will show how to convert this value to a voltage. The result of the measurement is then sent via the serial line by Serial.print() and Serial.println() commands. The difference between the two is that the former does not send a carriage return character at the end of the message, and the latter does. After the if() f g statement at the end of the loop, again a small delay is added.

I2C and SPI

I2C

The device has four pins—check the datasheet to find out which pin does what—for ground, supply voltage, and the I2C lines SDA as SCL. The latter two are connected to pin A4 and A5 on the UNO because the I2C data and clock lines are routed to the same output pins as the analog pins A4 and A5. These pins cannot be used for analog measurements, in case they are used for I2C connectivity.

```
// Read MLX90614 IRthermometer, V. Ziemann, 170717
#include <Wire.h>
const int MLX90614=0x5A; // I2C address
float getTemperature(uint8_t addr) { //.....getTemperature
uint16_t val;
uint8_t crc;
Wire.beginTransmission(MLX90614);
Wire.write(addr); // address
Wire.endTransmission(false);
Wire.requestFrom(MLX90614,3);
val = Wire.read();
val |= (Wire.read()<<8);
crc=Wire.read(); // not used
return -273.15+0.02*(float)val;
}
void setup() { //.....................................setup
Serial.begin(9600);
Wire.begin();
}
```

```
void loop() { //.....................................loop
float Ta=getTemperature(0x06); // address for ambient temp
float To=getTemperature(0x07); // address for object1 temp
Serial.print(Ta); Serial.print("\t"); Serial.println(To);
delay(1000);
}
```

First we have to include support for the I2C functionality by including the Wire.h header file, which also causes the compiler and linker to include the corresponding libraries. After we define the I2C address 0x5A of the sensor, which we find in the datasheet, we encapsulate the I2C communication in a separate function called getTemperature(). It receives the register address of the I2C device as input parameter and returns the associate temperature, properly scaled to degrees Celsius. The device also returns a byte that allows us to determine transmission errors, but we do not use that feature in this simple example. In the setup() function we only initialize Serial line and I2C communication. In the loop() function we use the getTemperature() function to retrieve the ambient temperature from address 0x06 and the object temperature from address 0x07, as described in the datasheet, and print both temperatures formatted with a tabulator to the Serial line. The entire process is then repeated after waiting 1000 ms.

SPI

Instead of bit banging the pins, we can also utilize the SPI library that comes with the Arduino IDE. The sketch, based on the ¡SPI.h¿ library, but otherwise equivalent to the previous one, is the following.

```
//␣MCP3304,␣V.␣Ziemann,␣170726
#include␣<SPI.h>
```

```
#define␣CS␣15␣//␣D8
int␣mcp3304_read_adc(int␣channel)␣{␣//...0␣to␣3.......read_adc
int␣adcvalue=0,␣b1=0,␣hi=0,␣lo=0,␣sign=0;;
digitalWrite␣(CS,␣LOW);
byte␣commandbits␣=␣B00001000;␣//␣Startbit+(diff=0)
commandbits␣|=␣channel␣&␣0x03;
SPI.transfer(commandbits);
b1␣=␣SPI.transfer(0x00);␣//␣always␣D0=0
sign␣=␣b1␣&␣B00010000;
hi␣=␣b1␣&␣B00001111;
lo␣=␣SPI.transfer(0x00);␣//␣input␣is␣don?t␣care
digitalWrite(CS,␣HIGH);
adcvalue␣=␣(hi␣<<␣8)␣+␣lo;
if␣(sign)␣{adcvalue␣=␣adcvalue-4096;}
return␣adcvalue;
}
void␣setup()␣{//.............................setup
pinMode(CS,OUTPUT);
SPI.begin();
SPI.setFrequency(2100000);
SPI.setBitOrder(MSBFIRST);
SPI.setDataMode(SPI_MODE0);
Serial.begin(115200);
while␣(!Serial)␣{yield();}
}
void␣loop()␣{␣//.........................loop
int␣val=mcp3304_read_adc(0);
```

```
Serial.print("CH0␣=␣");␣Serial.println(val);
delay(1000);
}
```

Here we first include the ¡SPI.h¿ functionality and define the CS pin before defining the mcp3304 read adc function to read a channel from the ADC. In this function we first declare a number of variables and pull the CS pin LOW in order to start the transaction. Then we build the commandbits; this time they are aligned such that the start bit is bit 3 and the channel information is stored in bits 1 and 0, and we use the SPI.transfer function to send it to the ADC. Note that the first bit recognized by the ADC is the first non-zero bit, which is the start bit 3. We then send 0x00 in a second call to SPI.transfer and receive

the readings from the MISO pin in b1. Since we have a number of idle clock toggles, the sign bit is bit 4 and the next four bits are the four most significant bits of the ADC reading, which we store in the variable hi. The next call to SPI.transfer() returns the subsequent bits, which are the lower eight bits from the MISO pin, and we store them in the variable lo. After pulling the CS pin HIGH to conclude the transaction, we build the ADC word adcvalue and add the sign information. Finally we return the reading to the calling program. In the setup() function we initialize the SPI functionality by calling the SPI.begin() function, setting the clock frequency, byte order, and MODE, and then initialize the serial line. The loop() function is a straight copy of the previous example, where we only read channel 0 and print the value to the serial line.

Interfacing Actuators

Switching devices

Dynamically adjusting the brightness of an LED is achieved by pulse-width modulation, and that feature is available on pins D3, D5, D6, D9, and D11 on the Arduino Uno. We use the analogWrite(pin,value) function that takes the pin number and a value between 0 and 255 (0 to 1023 on the ESP8266) to adjust the duty cycle of the pulse-width modulated signal from completely off to completely on. The switching frequency is between 500 and 1000 Hz, and in Figure 4.12, we show an oscilloscope trace of the result of analogWrite() to pin D9 of a UNO with values 88 and 220, respectively. The frequency of the signal is about 490 Hz, and we see the length of the signal differing from about 1/3 of the time at 5 V on the left to 5 V almost all the time on the right.

In order to be able to manipulate actuators in the same way as the sensors, we use the same query-response protocol to communicate the required actions to the microcontroller. We want to use the convention that DWx 0 turns digital pin number x off and DWx 1 turns it on, and that sending AWx nnn sets pulse-width modulation on pin number x to value nnn. The code that achieves this is the following:

```
// Switching_and_pwm, V. Ziemann, 170614
char line[30];
void setup() {
pinMode(2,OUTPUT);
Serial.begin (9600);
while (!Serial) {;}
}
void loop() {
if (Serial.available()) {
```

```
Serial.readStringUntil(?\n?).toCharArray(line,30);
if (strstr(line,"DW2 ")==line) {
int val=(int)atof(&line[4]);
digitalWrite(2,val);
} else if (strstr(line,"AW9 ")==line) {
int val=(int)atof(&line[4]);
analogWrite(9,val);
} else {
Serial.println("unknown");
}
}
}
```


DC motors

If we only want to control the speed of a very small motor, we can replace the LED and the 220 ? resistor in the previous example by the motor. We also need to add flyback diodes to prevent the back-emf from damaging the transistor. For slightly larger motors, we need to use a transistor with a higher power rating, such as a TIP-120 Darlington power transistor. The TIP-120 has a flyback diode from emitter to collector already built in. Note that the performance of larger transistors normally degrades at higher frequency. Inspection of the datasheet, however, shows that this only affects frequencies well above 10 kHz.

In Figure 4.14 we show the setup with a UNO controlling a small motor. The terminals of the transistor are base, collector, and emitter, from left to right, and the motor is connected between the collector and the supply voltage. The emitter is directly connected to ground, and we include an external flyback diode (horizontally mounted) between the motor leads, with the cathode pointing towards the right. The

vertically mounted diode illustrates the connection of the built-in diode. We control the speed of the motor by pulse width modulating the base of the transistor that is connected to pin D9 on the UNO via a 1 k? resistor.

```
// H bridge DC motor Controller, V. Ziemann, 170614
char line[30];
void setup() {
pinMode(2,OUTPUT);
pinMode(3,OUTPUT);
Serial.begin (9600);
while (!Serial) {;}
}
void loop() {
if (Serial.available()) {
Serial.readStringUntil(?\n?).toCharArray(line,30);
if (strstr(line,"FW ")==line) {
digitalWrite(2,LOW);
digitalWrite(3,LOW);
digitalWrite(3,HIGH);
float val=atof(&line[3]);
analogWrite(9,(int)val);
} else if (strstr(line,"BW ")==line) {
digitalWrite(2,LOW);
digitalWrite(3,LOW);
digitalWrite(2,HIGH);
float val=atof(&line[3]);
analogWrite(9,(int)val);
} else { // STOP in all other cases
```

```
digitalWrite(2,LOW);
digitalWrite(3,LOW);
analogWrite(9,(int)0);
}
}
}
```

Lecture 5

Raspberry Pi I

The Raspberry Pi7 is a series of single-board computers created in the United Kingdom by the Raspberry Pi Foundation. The original aim of the founders was to encourage the teaching of basic computer science in schools and developing countries. Step by step, their boards significantly changed the way manufacturers and developers thought about and created new projects in many application scenarios. For example, the original model became very popular and started spreading outside of its initial target market for uses such as robotics.

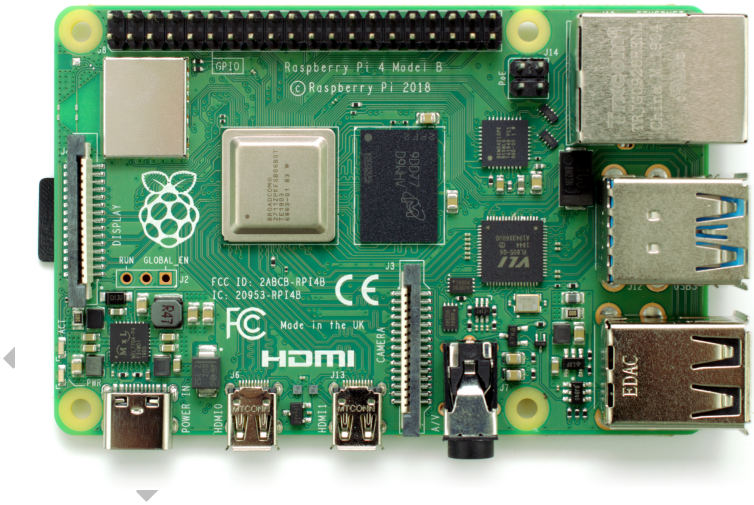
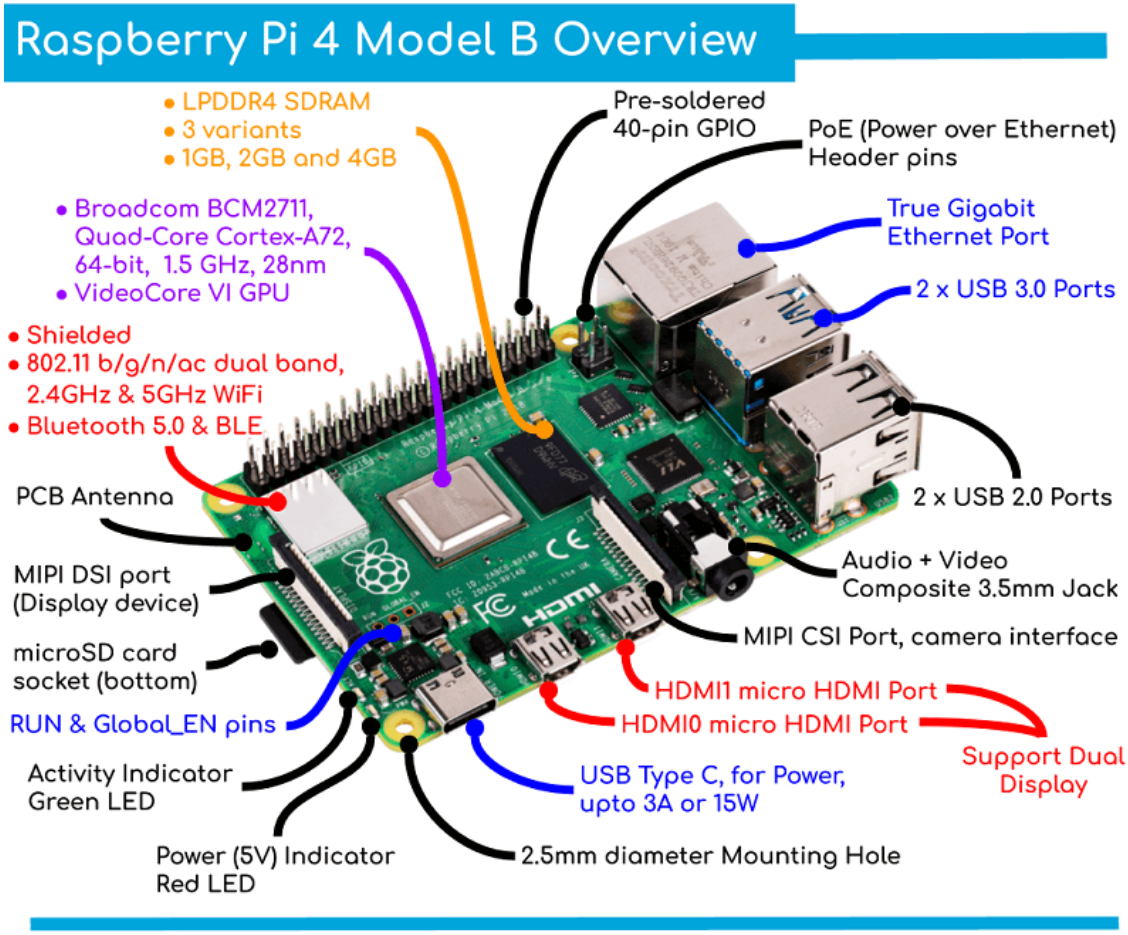


Figure 7.7 shows three Raspberry Pi boards with their available components and hardware profiles. The Broadcom BCM2835 SoC was used on the first generation and was inspired by the chip used in first-generation smartphones (its CPU is an older ARMv6 architecture). It includes a 700 MHz ARM1176JZF-S processor, VideoCore IV GPU, and RAM. It has a level 1 (L1) cache of 16 kB and a level 2 (L2) cache of 128 kB. The level 2 cache is used primarily by the GPU. The SoC is stacked underneath the RAM chip, so only its edge is visible.

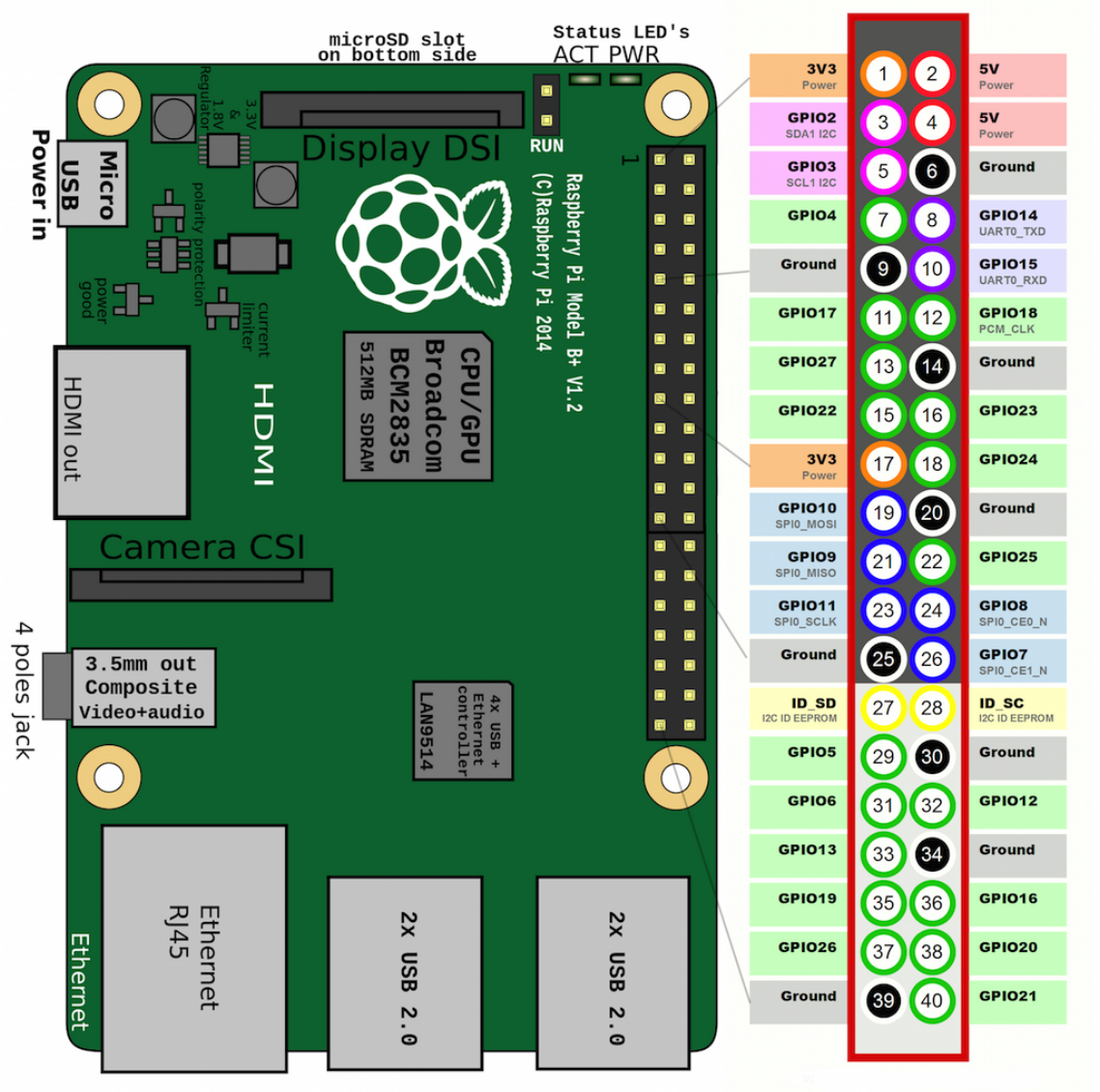
The Raspberry Pi 2 uses a Broadcom BCM2836 SoC with a 900 MHz 32-bit quad-core ARM Cortex-A7 processor (as do many current smartphones), with 256 kB shared L2 cache. The Raspberry Pi 3 uses a Broadcom BCM2837 SoC with a 1.2 GHz 64-bit quad-core ARM Cortex-A53 processor, with 512 kB shared L2 cache.



The Model A, A+ and Pi Zero are shipped without Ethernet modules and are commonly connected to a network using external adapters for Ethernet or Wi-Fi. Models B and B+ have the Ethernet port that is provided by a built-in USB Ethernet adapter using the SMSC LAN9514 chip. The Raspberry Pi 3 and Pi Zero W (wireless) provide a 2.4 GHz Wi-Fi 802.11n (150 Mbit/s) and Bluetooth 4.1 (24 Mbit/s) connectivity module based on a Broadcom BCM43438 chip. The Raspberry

Pi 3 is also equipped with a 10/100 Ethernet port.

The Raspberry Pi can be also used with USB storage, USB-to-MIDI converters, and virtually any other device/component with USB capabilities. Other external devices, sensors/actuators and peripherals can be attached through a set of pins and connectors available on the board?s surface.



The Raspberry Pi board family can run multiple operating systems, such as Raspbian, Fedora, Ubuntu MATE, Kali Linux, Ubuntu Core, Windows 10 IoT Core,

RISC OS, Slackware, Debian, Arch Linux ARM, and Android Things. This combination of high-profile hardware, software and operating systems makes these boards to represent powerful and complex nodes in heterogenous IoT applications. They can efficiently work as IoT Hubs, gateways and data collectors using heterogenous protocols and running multiple services at the same time. All the Raspberry Pi boards can be classified as Class 2 constrained devices.

But beyond the CPU, the Raspberry Pi sports a video processor that can display videos at full-HD resolution (1920×1080) via the built-in HDMI-connector. Audio output is available either via the HDMI connector or via a 3.5-mm headphone connector. Moreover, there are four USB-2 ports on board to connect peripheral components, such as USB sticks, keyboard, mice, or web cameras. Communication with the outside world is feasible via a built-in wired Ethernet port, and since version 3, the Raspi has had built-in Bluetooth (V4.1) and WiFi (802.11n).

The Raspis are very attractive due to their built-in low-level peripherals. There are 17 general-purpose inputoutput (GPIO) pins exposed on the board, some of which support I2C, SPI, and UART (RS-232-like) communication. Moreover, a specific audio bus (I2S) is available, as well as a high-speed CSI interface to connect the tailor-made Raspberry Pi camera, and a DSI interface to connect LCD panels.

Lecture 6

Raspberry Pi II - Application

RASPI AS A ROUTER

On the Raspi we need to install the following packages

```
sudo apt-get update
sudo apt-get install hostapd dnsmasq
```

where we first update the repositories, and then download and install the hostapd and dnsmasq package. The former program is responsible for spanning the private wireless network, and turns the Raspi into a WLAN access point. The latter provides IP numbers on the private network via the DHCP protocol, and translates web site names to IP numbers. Next we need to re-configure our system. In the default configuration there is a background process, a daemon called dhcpd, that tries to obtain IP numbers for every network interface on the Raspi, including the wireless interface called wlan0. On the other hand, in order to operate the Raspi as an access point, we need to configure the interface ourselves and therefore need to remove the interface wlan0 from the control of the daemon by adding the line

```
denyinterfaces wlan0
```

at the end of the file /etc/dhcpcd.conf with any text editor, such as nano or emacs. Now we are ready to configure the interface wlan0 manually by rewriting the relevant section in the file /etc/network/interfaces to look like this:

```
auto wlan0
iface wlan0 inet static
address 192.168.20.1
```

```
netmask 255.255.255.0
up /sbin/iptables -A POSTROUTING -t nat -o eth0 -j MASQUERADE
down /sbin/iptables -D POSTROUTING -t nat -o eth0 -j MASQUERADE
```

which defines the IP number of the wlan0 interface to be 192.168.20.1, and configures the type of network (class C) in the following line with the netmask command. The /sbin/iptables command enables network address translation between the wireless network on wlan0 and the wired network on interface eth0. In order to allow network packages to pass between the interfaces, which is disabled by default as a security measure, we need to enable forwarding by removing the hash from the line

```
net.ipv4.ip_forward=1
```

in the file /etc/sysctl.conf. We omit this step if all sensor nodes should only communicate with the Raspi, but no other computer on the local network should be able to surf the outside Internet beyond the Raspi, which might not be needed and poses a potential security risk. This last step completes the basic network setup, and we turn to the configuration of the access point software.

The configuration file for the hostapd daemon is /etc/hostapd/hostapd.conf and contains the following lines:

```
# /etc/hostapd/hostapd.conf
interface=wlan0
driver=nl80211
ssid=messnetz
channel=5
macaddr_acl=0
auth_algs=1
ignore_broadcast_ssid=0
wpa=2
```

```
wpa_passphrase=zxcvZXCV
wpa_key_mgmt=WPA-PSK
wpa_pairwise=TKIP
rsn_pairwise=CCMP
hw_mode=g
```

The file must be made readable for the owner only by executing sudo chmod 600 /etc/hostapd/hostapd.conf. It contains the definition of the network name messnetz and the details of the encryption, such as the passphrase. The latter we should adapt to your own secret phrase, to prevent others from misusing your wireless network. We need to tell the system where to find the configuration file by entering the line

```
DAEMON_CONF=/etc/hostapd/hostapd.conf
```

near the top of the file /etc/init.d/hostapd and then instruct the system to start the hostapd daemon at boot time with the command.

Communicating with Arduino

From Commandline

At this point we have established the Raspi as a development system for Arduino software, and ensure it communicates via the USB-based serial line with the Arduino IDE. But normally we want to access the Arduino from self-written programs rather than always using the Arduino IDE. The first option is to use the screen program or any other terminal program to connect to the UNO. After closing the Arduino IDE, we therefore start the following program on the command line on the Raspi:

```
screen /dev/ttyACM0 9600
```

just as we did earlier when the UNO was connected to the Desktop computer. Bluetooth communication works exactly the same way as on the desktop computer.

Python

We use the following Python script to query the UNO with the query-response sketch for a single value from analog pin 0.

```
# query_arduino.py
import serial, time
query="A0?\n"
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)
time.sleep(1) # wait for serial to be ready
ser.write(query)
time.sleep(0.1)
reply=ser.readline()
print(reply.strip())
ser.close()
```

Python is rather lightweight, and we have to import extra functionality such as serial communication by executing

```
sudo apt-get install python-serial
```

t the command prompt and then load the new functionality with the import statement. In the above script we import support for handling the serial line and basic time-handling. The latter we need to implement delays in the script. In the second line we define a variable query that contains the query string we send to the Arduino. Note that we explicitely add the carriage return character. Then we open the serial port ser on /dev/ttyACM0 with a baud rate of 9600 and a timeout of 1 second, which prevents unsatisfied read attempts from blocking the program. We wait for one second to allow the operating system to finish opening the serial port, and then submit the query string with the ser.write command. Note that we use the method

write on the serial device ser. Then we wait for 0.1 seconds and read characters up to the CR-LF character with the ser.readline() function into the variable reply. We display the reply with the print command after stripping off leading and trailing white-space characters, such as normal spaces or CR-LF characters. Finally, we close the serial line.

Data Storage

Flatfile

The simplest database is certainly a file containing a time stamp and one or more measurement values, possibly even in human-readable form. This is called a flatfile database, and we create one by using the following Python script:

```
# logger with time, V Ziemann, 161228
import serial, time, sys, atexit
def cleanup():
ser.close()
atexit.register(cleanup)
query="A0?\n"
ser=serial.Serial("/dev/ttyACM0",9600,timeout=1)
time.sleep(1) # wait for serial to be ready
while 1:
ser.write(query)
time.sleep(0.1)
reply=ser.readline()
print int(time.time()), reply[3:].strip()
sys.stdout.flush()
time.sleep(1)
```

and store it in a file called ask repeat.py. We recognize the same organization as before, when we discussed Python scripts with importing functionality, registering the cleanup() function, opening the serial line, and repeatedly sending the query string and receiving the reply. The only difference is that we print the Unix time, which is the number of seconds since January 1, 1970, also called the epoch, before the measurement value.

Lecture 7

Real Time Operating System I

OpenWSN

The OpenWSN8 project is an open-source implementation of a fully standards-based protocol stack for IoT networks. It was based on the new IEEE802.15.4e time-slotted channel-hopping standard. IEEE802.15.4e, coupled with IoT standards such as 6LoWPAN, RPL and CoAP, enables ultra-low-power and highly reliable mesh networks that are fully integrated into the Internet.

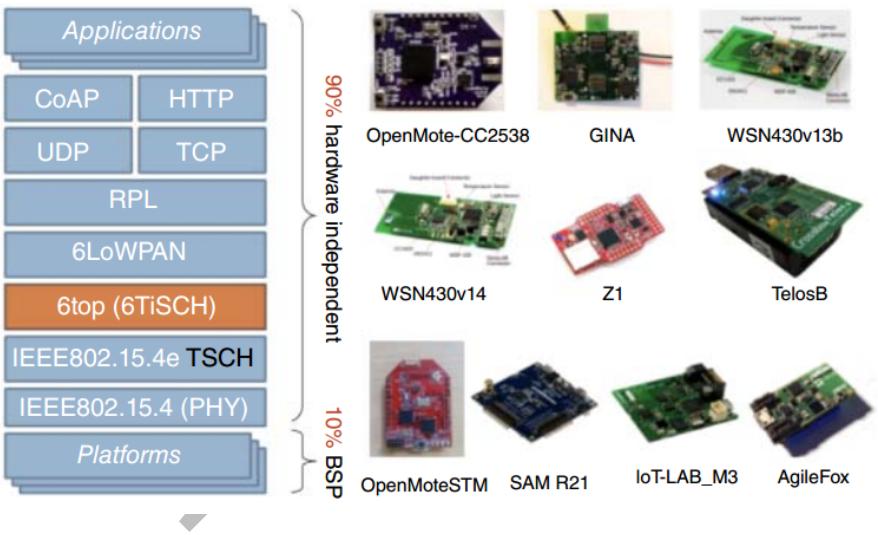


Figure 1: OpenWSN protocol stack, highlighting hardware-independent modules and supported hardware platforms.

OpenWSN has been ported to numerous commercial available platforms from older 16-bit micro-controllers to state-of-the-art 32-bit Cortex-M architectures. The OpenWSN project offers a free and open-source implementation of a protocol stack and the surrounding debugging and integration tools, thereby contributing to the

overall goal of promoting the use of low-power wireless mesh networks. Figure 7.8 shows the OpenWSN protocol layers and software libraries, which are hardware independent, and a set of supported hardware platforms, on which it can be installed and used.

TinyOS

TinyOS9 is a free, open-source, BSD-licensed OS designed for lowpower embedded distributed wireless devices used in sensor networks. It has designed to support the intensive concurrent operations required by networked sensors, with minimal hardware requirements. TinyOS was developed by University of California, Berkeley, Intel Research, and Crossbow Technology. It is written in the nesC(Network Embedded Systems C) programming language, which is a version of C optimized to support components and concurrency. It is also component-based, supporting event-driven programming of applications for TinyOS.

FreeRTOS

FreeRTOS10 is a real-time operating system kernel for embedded devices designed to be small and simple. It been ported to 35 micro-controllers and it is distributed under the GPL with an optional exception. The exception permits users? proprietary code to remain closed source while maintaining the kernel itself as open source, thereby facilitating the use of FreeRTOS in proprietary applications.

In order to make the code readable, easy to port, and maintainable, it is written mostly in C, (but some assembly functions have been included to support architecture-specific scheduler routines). It provides methods for multiple threads or tasks, mutexes, semaphores and software timers.

TI-RTOS

TI-RTOS11 is a real-time operating system that enables faster development by eliminating the need for developers to write and maintain system software such as schedulers, protocol stacks, power-management frameworks and drivers. It is provided with full C source code and requires no up-front or runtime license fees. TI-RTOS scales from a low-footprint, real-time preemptive multitasking kernel to a complete RTOS with additional middleware components including a power manager, TCP/IP and USB stacks, a FAT file system, and device drivers, allowing developers to focus on differentiating their applications.

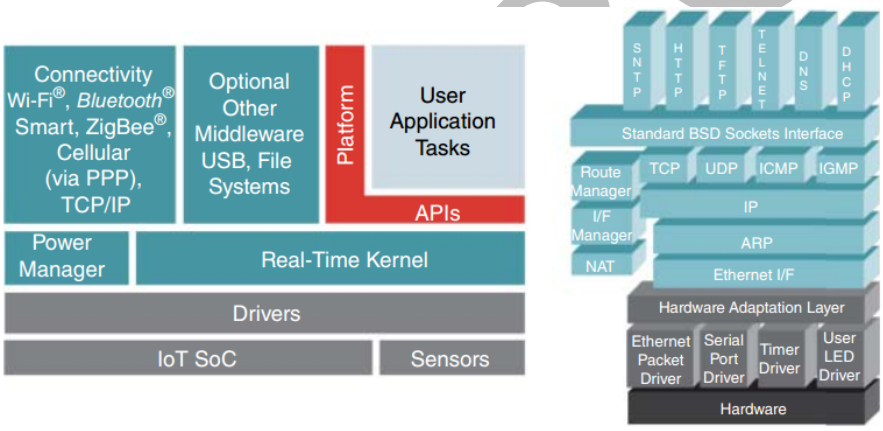


Figure 2: Schematic overview of the TI-RTOS operating system with main modules and software components.

Figure shows the main software components of the TI-RTOS operating system. In particular, it is based on a core layer with a real-time kernel, connectivity support and power management. On top of that, a set of platform APIs allow the developer to build custom applications. The OS provides a large set of ready-to-use libraries based on TCP/UDP/IP networking, standard BSD socket interface and main application layer protocols such as HTTP, TFTP, Telnet, DNS, and DHCP.

RIOT

RIOT12 is an open-source microkernel operating system for the IoT, licensed as LGPL. It allows C and C++ application programming, and provides both full multi-threading and real-time capabilities (in contrast to other operating systems with similar memory footprints, such as TinyOS or Contiki). RIOT runs on 8-bit (e.g., AVR Atmega), 16-bit (e.g., TI MSP430) and 32-bit hardware (e.g., ARM Cortex). A native port also enables RIOT to run as a Linux or MacOS process, enabling the use of standard development and debugging tools such as GNU Compiler Collection, GNU Debugger, Valgrind, Wireshark, an so on. RIOT is partly POSIX-compliant and provides multiple network stacks, including IPv6, 6LoWPAN and standard protocols such as RPL, UDP, TCP, and CoAP.

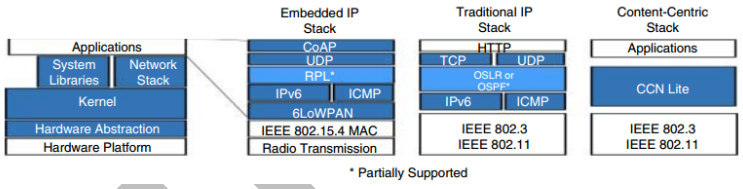


Figure 3: Overview of networking architecture for the RIOT operating system.

Lecture 8

RTOS II

Now in this section we will see special OS for IoT - Contiki

Contiki OS

Contiki is an operating system for networked, memory-constrained systems, targeting low-power wireless IoT devices. Its main characteristics are:

- It is open source and in continuous development. Even if it is less well documented and less well maintained than commercial operating systems, it allows developers not only to work on custom applications but also to modify core OS functionalities such as the TCP/IP stack and the routing protocol
- It provides a full TCP/uIPv6 stack using 6LoWPAN for header compression, and creates LR-WPAN routes with RPL, the IPv6 routing protocol for low-power and lossy networks.

Contiki was created by Adam Dunkels in 2002 and has been further developed by a worldwide team of developers from Texas Instruments, Atmel, Cisco, ENEA, ETH Zurich, Redwire, RWTH Aachen University, Oxford University, SAP, Sensinode, the Swedish Institute of Computer Science, ST Microelectronics, Zolertia, and many others.

Contiki is designed to run on classes of hardware devices that are severely constrained in terms of memory, power, processing power, and communication bandwidth. For example, in terms of memory, despite providing multitasking and a built-in TCP/IP stack, Contiki only needs about 10 kB of RAM and 30 kB of ROM. A typical Contiki system has memory of the order of kilobytes, a power budget of the

order of milliwatts, processing speed measured in megahertz, and communication bandwidth of the order of hundreds of kilobits/second. This class of systems includes various types of embedded systems as well as a number of old 8-bit computers.

A brief description of the core features of Contiki will be provided, highlighting why they are of particular interest for building complex IoT applications.

Networking

Contiki provides three network mechanisms:

- the uIP13 TCP/IP stack, which provides IPv4 networking;
- the uIPv6 stack, which provides IPv6 networking
- the Rime stack, which is a set of custom lightweight networking protocols designed specifically for low-power wireless networks.

The IPv6 stack was contributed by Cisco and was, at the time of release, the smallest IPv6 stack to receive IPv6-ready certification. The IPv6 stack also contains the RPL routing protocol and the 6LoWPAN header compression and adaptation layer.

The Rime stack is an alternative network stack that is intended to be used when the overhead of the IPv4 or IPv6 stacks is prohibitive. The Rime stack provides a set of communication primitives for lowpower wireless systems. The default primitives are single-hop unicast, single-hop broadcast, multi-hop unicast, network flooding, and address-free data collection. The primitives can be used on their own or combined to form more complex protocols and mechanisms.

Low-Power Operation

Many Contiki systems are severely power-constrained. Battery operated wireless sensors may need to provide years of unattended operation, often with no way to

recharge or replace batteries. Contiki provides a set of mechanisms for reducing the power consumption of the system on which it runs. The default mechanism for attaining low-power operation of the radio is called ContikiMAC. With ContikiMAC, nodes can be running in low-power mode and still be able to receive and relay radio messages.

Simulation

The Contiki system includes a network simulator called Cooja . Cooja simulates networks of Contiki nodes. The nodes may belong to one of three classes:

- emulated nodes, where the entire hardware of each node is emulated;
- Cooja nodes, where the Contiki code for the node is compiled and executed on the simulation host;
- Java nodes, where the behavior of the node must be reimplemented as a Java class.

A single Cooja simulation may contain a mixture of nodes from any of the three classes. Emulated nodes can also be used, so as to include on-Contiki nodes in a simulated network.

In Contiki 2.6, platforms with TI MSP430 and Atmel AVR microcontrollers can be emulated. Cooja can be very useful because of its emulative functions, which help developers in testing applications. This speeds up the development process: without a simulator the developer would have to upload and test every new version of firmware on real hardware. This would be a long process, because most motes can only be flashed through a (slow) serial port.

Programming Model

To run efficiently on memory-constrained systems, the Contiki programming model is based on protothreads. A protothread is a memory-efficient programming abstraction that shares features of both multi-threading and event-driven programming to attain a low memory overhead. The kernel invokes the protothread of a process in response to an internal or external event. Examples of internal events are timers that fire, or messages being posted from other processes. Examples of external events are sensors that trigger, or incoming packets from a radio neighbor.

Protothreads are cooperatively scheduled. This means that a Contiki process must always explicitly yield control back to the kernel at regular intervals. Contiki processes may use a special protothread construct to avoid waiting for events while yielding control to the kernel between each event invocation.

Features

ontiki supports per-process optional preemptive multi-threading, inter-process communication using message-passing events, and an optional GUI subsystem with either direct graphic support for locally connected terminals or networked virtual display with virtual network computing or over Telnet.

A full installation of Contiki includes the following features:

- multitasking kernel
- optional per-application pre-emptive multithreading
- protothreads
- TCP/IP networking, including IPv6
- windowing system and GUI

- networked remote display using virtual network computing
- web browser (claimed to be the world?s smallest)