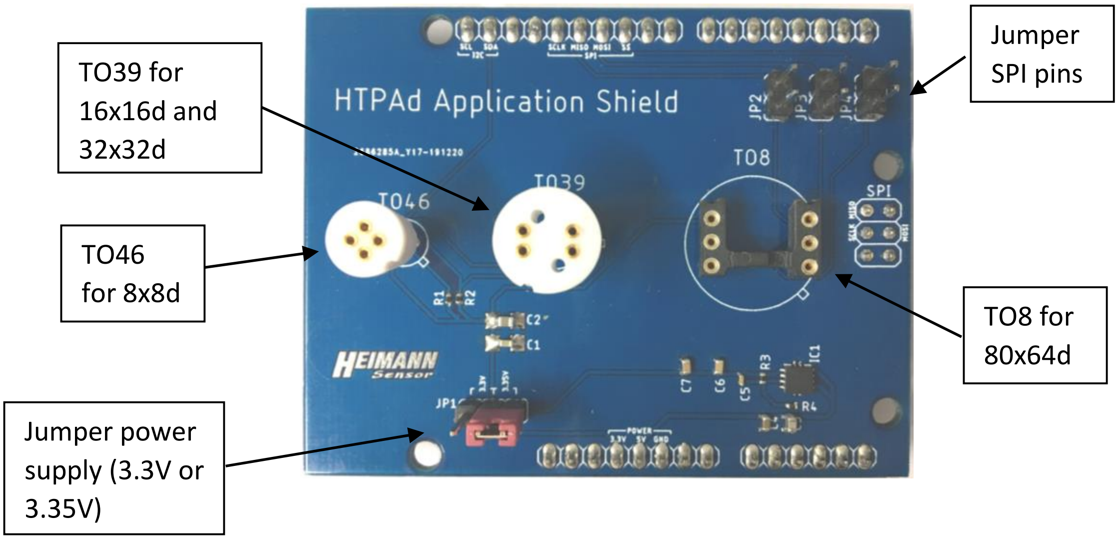
Content

Content 1
1 Overview 2
  1.1 Serial Mode 3
  1.2 Ethernet Mode 4
2 Installation 5
  2.2 Upload 9
3 Power supply 10
4 Sample Code 11
  4.1 Serial mode 11
    4.1.1 setup 11
    3.1.2 loop 13
  3.2 Ethernet mode 15
    3.2.1 setup 15
    3.2.2 IP and subnet address 16
    3.2.3 loop 17

1 Overview

The HTPAd Application Shield was created to read thermopile arrays with a STM32 F446RE Nucleo-64 board. The PCB support the I2C sensors: 8x8d, 16x16d and 32x32d and the SPI sensor 80x64d and 120x84d. The PCB can be used in two modes: Serial Mode and Ethernet Mode. In the Serial Mode the Nucleo board prints the temperature values in the Serial Monitor of the Arduino IDE. In Ethernet Mode you can stream temperature pictures continuously with the GUI (Heimann Sensor ArraySoft v2).



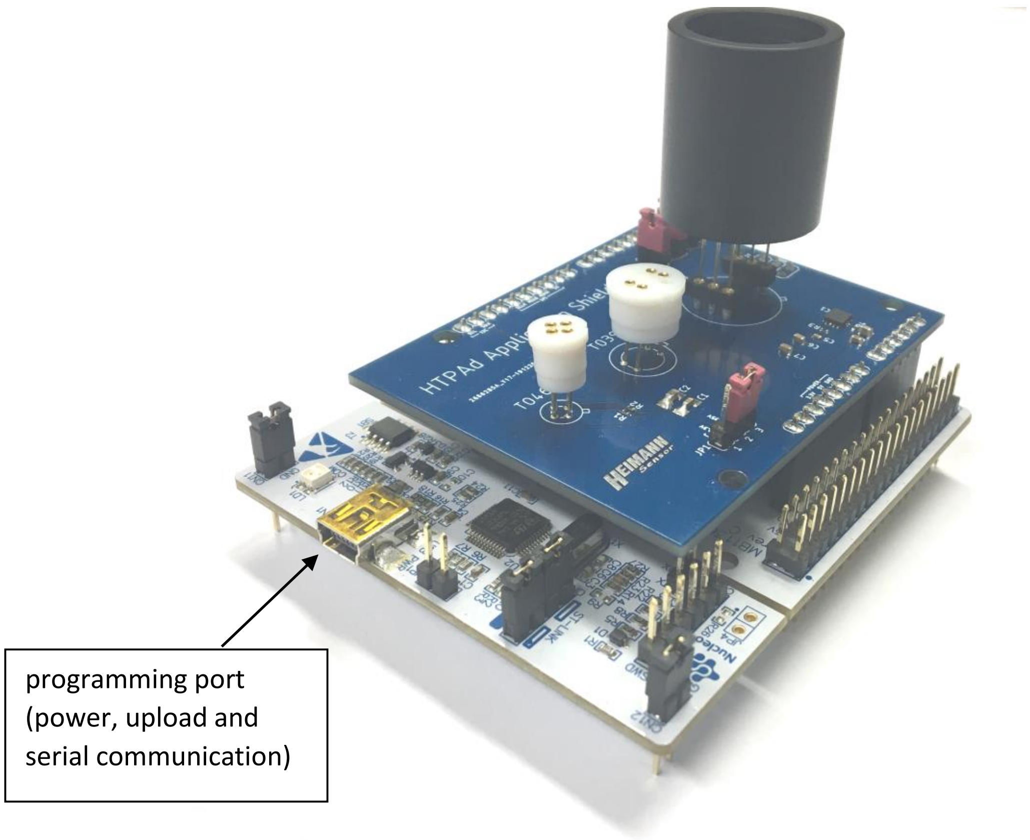
HINT: You cannot use more than one I2C sensor simultaneously with the PCB. The EEPROM of the sensors could be overwritten at a wrong place, if you connect more than one sensor. This could make your sensors unusable.

Some Arduinos have a SPI interface on pin 10-13 and on the 6-Pin SPI header. The Arduino Due only has these 6 pin header. For other boards, like Arduino Uno or STM32 Nucleo, you have to bridge the jumpers JP2, JP3 and JP4. These jumpers connect pin 10-13 with the sensor. The Arduino Due doesn’t need them.

The sensors can be powered directly via 3.35V. With the jumper JP1 you can switch between 3.3V and 3.35V as VDD.

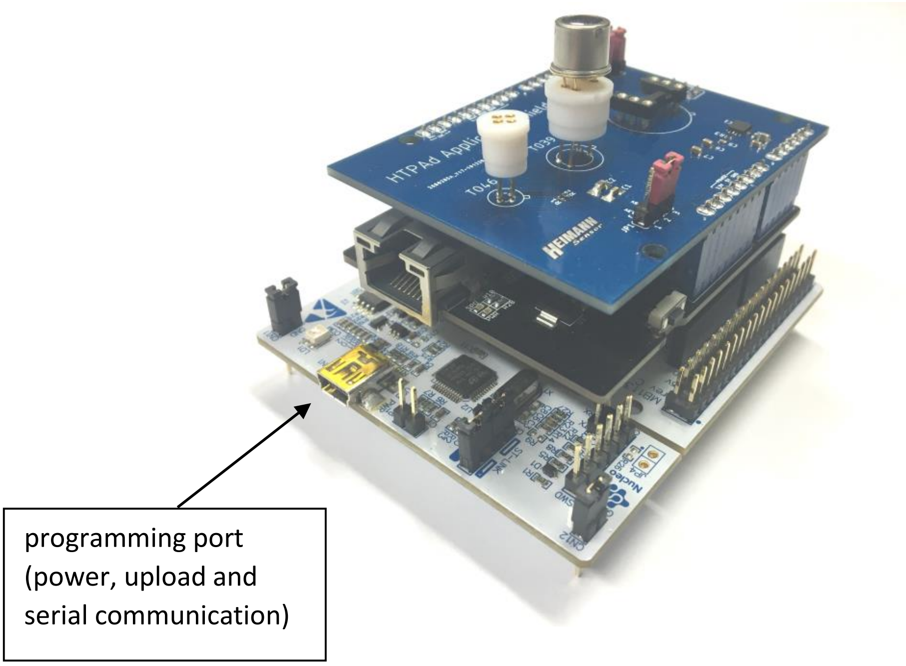
1.1 Serial Mode

- for all array types (8x8d, 16x16d, 32x32d, 80x64d and 120x84d)
- output in Serial Monitor of Arduino IDE
- required hardware: STM32F446RE Nucleo
- functions:
    - check EEPROM content
    - print picture with pixel temperatures
    - print all calculation steps



1.2 Ethernet Mode

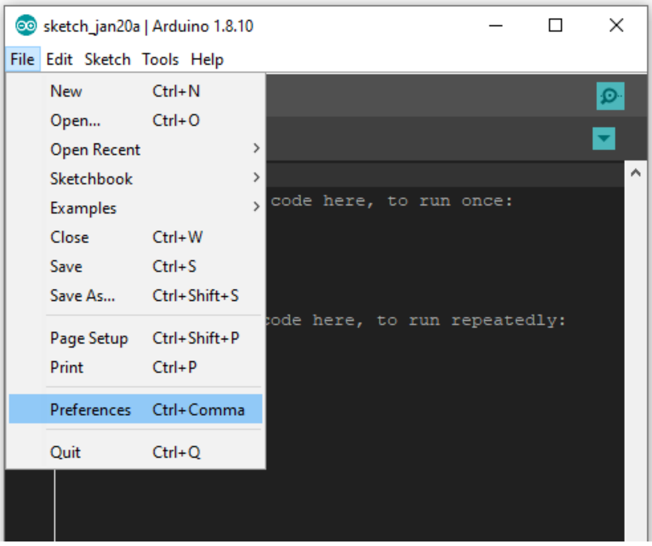
- only for I2C sensors (8x8d, 16x16d, 32x32d)
- output in GUI (Heimann Sensor ArraySoft v2)
- required hardware: STM32F446RE Nucleo and Ethernet Shield
- functions:
    - false color visualization of images
    - stream continuously
    - interpolation
    - temperature or voltage picture
    - record stream as txt/BDS



2 Installation

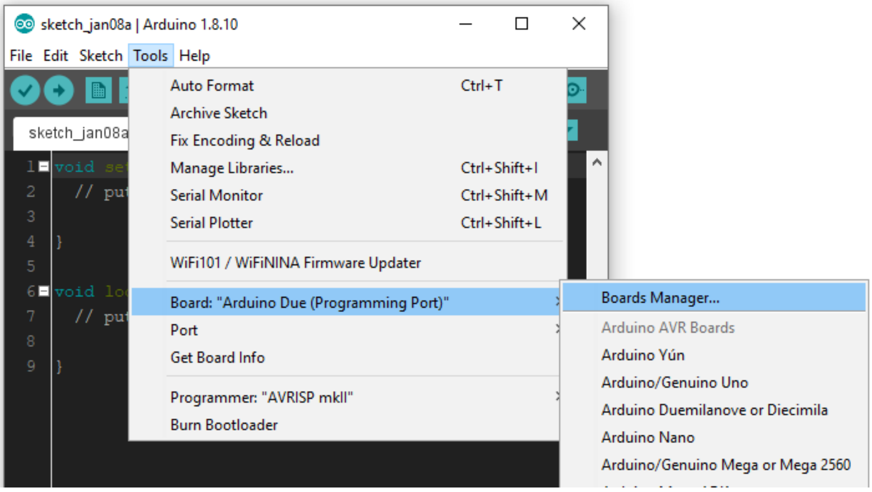
Follow these steps to start with Arduino Due:

1. Download and install the Arduino IDE from: https://www.arduino.cc/en/Main/Software
2. Open the Arduino IDE and click on File -> Preferences

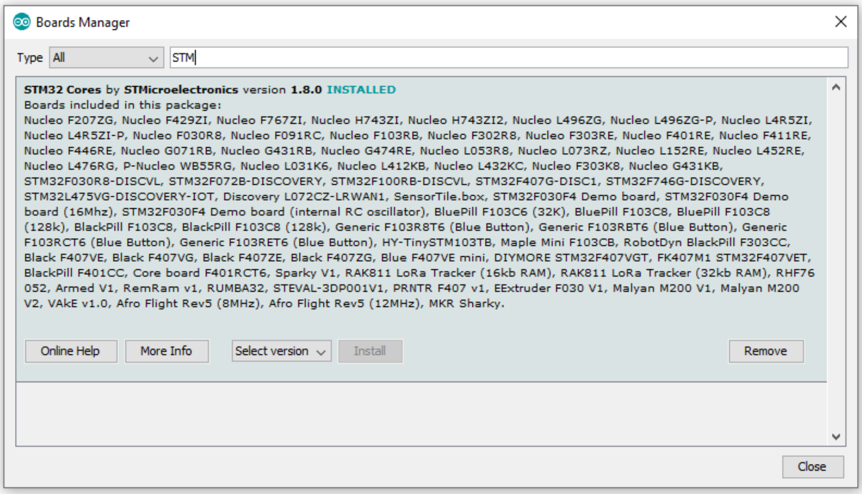


3. Copy the link to "Additional Boards Manager URLs:" and click OK: https://github.com/stm32duino/BoardManagerFiles/raw/master/STM32/package_stm_index.json

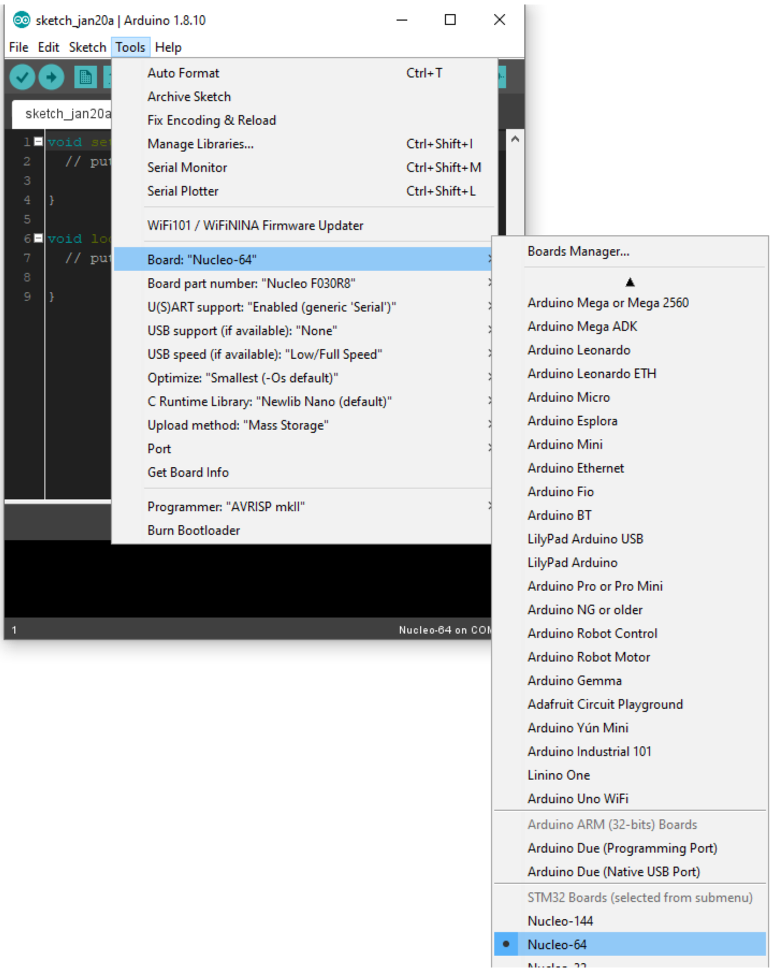
4. You have to add the STM32 Nucleo boards:
   a. Open the board manager (Tools -> Board -> Boards Manager...)



b. Install **STM32 Cores**



The Nucleo boards should appear in your list.



5. To upload a program, you have to change the following settings:
   Tools -> Board: Nucleo-64
   Tools -> Board part number: F446RE
   Tools -> Upload method: STMCubeProgrammer (SWD)

   Maybe an error appears during the first upload. The error message includes a link to install for STM Cube Programmer. Follow the instructions to install this programmer. We strongly recommend to install into the default folder, since in our testing setup numerous problems were created if installed into a different directory.

6. Only for I2C devices:
   The installation of the Nucleo board adds a new library for the I2C: **Wire.h**. The installed Wire.h library only allows to receive data with a length of 32 bytes. The maximal length of an I2C packet is 258 bytes (block of 32x32d). Therefore, changes in Wire.h and Wire.cpp are required. Better copy the changed wire library from the cd and replace the installed library in your folder.
   Example path: C:\Users\NAME\AppData\Local\Arduino15\packages\STM32\hardware\stm32\1.8.0\libraries\Wire\src

Changes in Wire.h (ver.1.8.0)

line	old	new
32	#define BUFFER_LENGTH 32	#define BUFFER_LENGTH 258
41	static uint8_t rxBufferAllocated;	static uint16_t rxBufferAllocated;
42	static uint8_t rxBufferIndex;	static uint16_t rxBufferIndex;
43	static uint8_t rxBufferLength;	static uint16_t rxBufferLength;
97	uint8_t requestFrom(uint8_t, uint8_t);	uint8_t requestFrom(uint8_t, uint16_t);
98	uint8_t requestFrom(uint8_t, uint8_t, uint8_t);	uint8_t requestFrom(uint8_t, uint16_t, uint8_t);
99	uint8_t requestFrom(uint8_t, uint8_t, uint32_t, uint8_t, uint8_t);	uint8_t requestFrom(uint8_t, uint16_t, uint32_t, uint8_t, uint8_t);

Changes in Wire.cpp (ver.1.8.0)

line	old	new
32	uint8_t TwoWire::rxBufferAllocated = 0;	uint16_t TwoWire::rxBufferAllocated = 0;
33	uint8_t TwoWire::rxBufferIndex = 0;	uint16_t TwoWire::rxBufferIndex = 0;
34	uint8_t TwoWire::rxBufferLength = 0;	uint16_t TwoWire::rxBufferLength = 0;
130	uint8_t read = 0;	uint16_t read = 0;

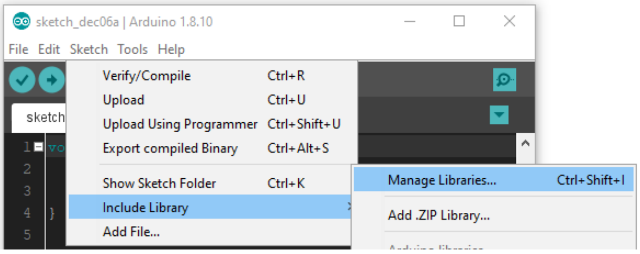
180-183	uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop)
	{
	return requestFrom((uint8_t)address, (uint8_t) quantity, (uint32_t)0, (uint8_t)0, (uint8_t)sendStop);
	}
		uint8_t TwoWire::requestFrom(uint8_t address, uint16_t quantity, uint8_t sendStop)
		{
		return requestFrom((uint8_t)address, (uint16_t) quantity, (uint32_t)0, (uint8_t)0, (uint8_t)sendStop);
		}

185-188	uint8_t TwoWire::requestFrom(uint8_t address, uint8_t quantity)
	{
	return requestFrom((uint8_t)address, (uint8_t) quantity, (uint8_t)true);
	}
		uint8_t TwoWire::requestFrom(uint8_t address, uint16_t quantity)
		{
		return requestFrom((uint8_t)address, (uint16_t) quantity, (uint8_t)true);
		}

190-193	uint8_t TwoWire::requestFrom(int address, int quantity)
	{
	return requestFrom((uint8_t)address, (uint8_t) quantity, (uint8_t)true);
	}
		uint8_t TwoWire::requestFrom(int address, int quantity)
		{
		return requestFrom((uint8_t)address, (uint16_t) quantity, (uint8_t)true);
		}

195-198	uint8_t TwoWire::requestFrom(int address, int quantity, int sendStop)
	{
	return requestFrom((uint8_t)address, (uint16_t) quantity, (uint8_t)sendStop);
	}
		uint8_t TwoWire::requestFrom(int address, int quantity, int sendStop)
		{
		return requestFrom((uint8_t)address, (uint8_t) quantity, (uint8_t)sendStop);
		}

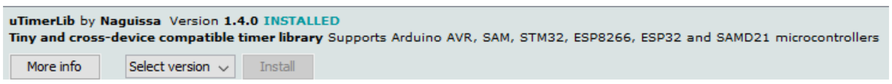
7. Only for Ethernet Mode:
   Install the libraries **Ethernet.h** and **uTimerLib.h**
   a. Open the library manager (Sketch -> Include Library -> Manage Libraries)



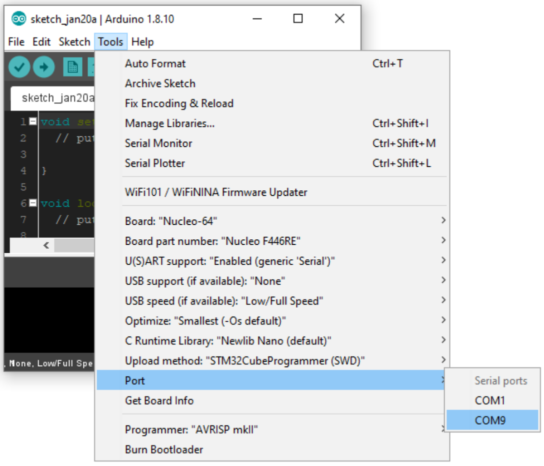
   b. Install **Ethernet.h**

   Hint: The Ethernet.h library defines a response time of 60 seconds. During this time the Arduino/Nucleo try to connect with a DCHP. If you are not using a DCHP and don't want to wait 60 seconds, change *timeout* and *responseTimeout* in Ethernet.h.

   c. Install **uTimerLib.h**



8. Restart the Arduino IDE and select the right port (Tools -> Port). For STM32 it is the port with the highest number.

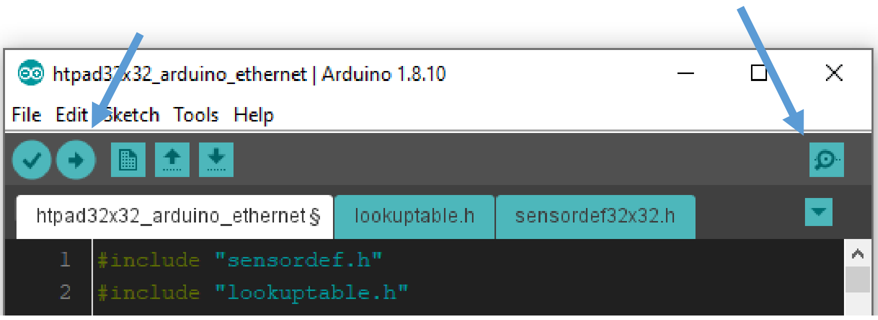


2.2 Upload

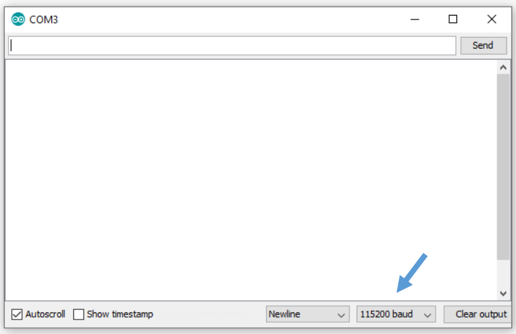
The folder contains three files (here for 32x32d):

- **htpad32x32_nucleo_ethernet.ino** or **htpad32x32_nucleo_serial.ino**
  (Sample code; Includes all functions to read the Sensor/EEPROM, calculate the temperature picture and communicate with GUI or Serial Monitor)
- **lookuptable.h**
  (includes the lookup tables for all in Sensordef32x32.h defined sensors)
- **sensordef_32x32.h**
  (define sensor and EEPRON addresses, sensor types, …)

Choose the right folder for your sensor and double-click the ino-file. Here you can upload the program and check the inputs with the Serial Monitor.

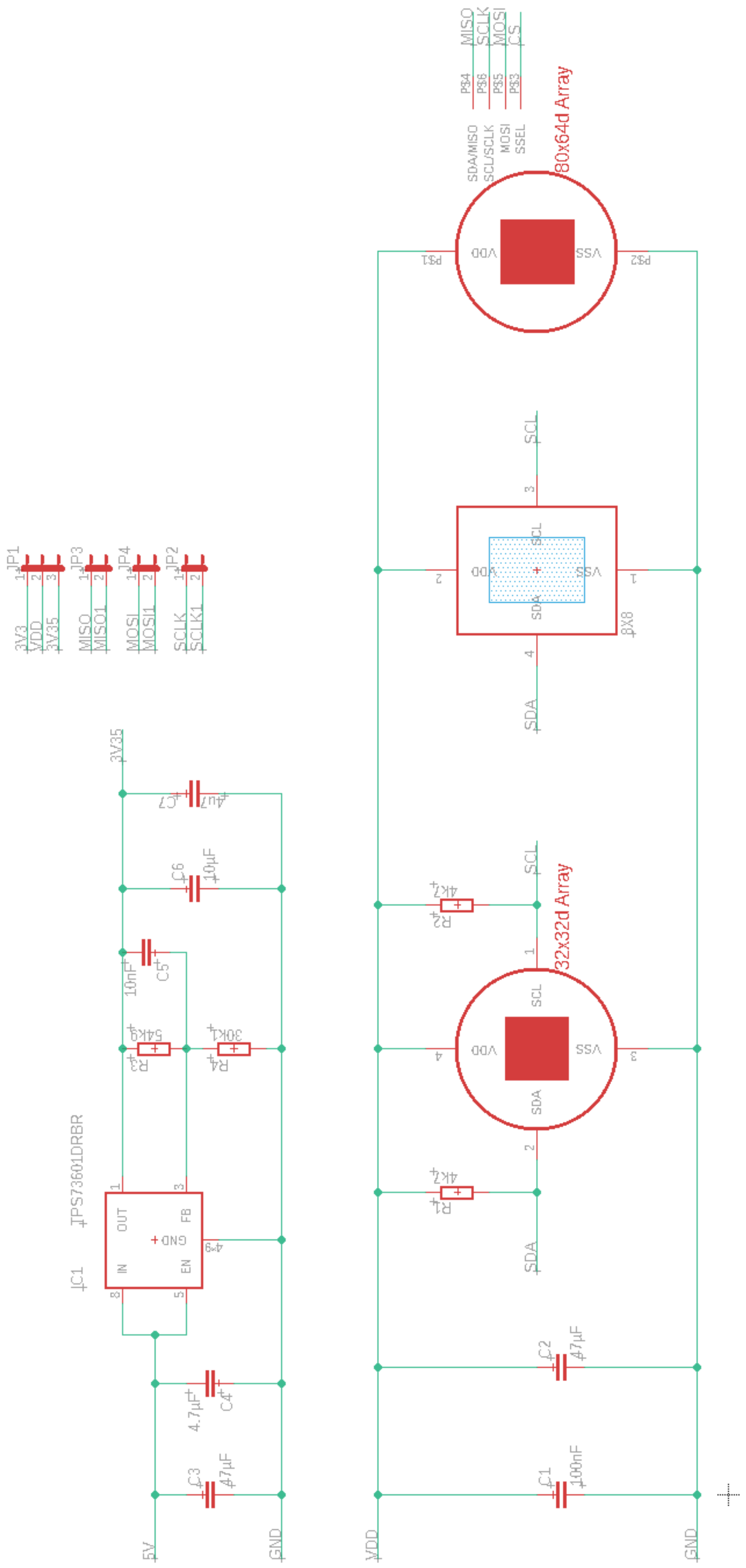


Therefore, you have to set the baud rate to 115200.



3 Power supply

The Sensors can be powered directly via 3.35V. To generate 3.35V the following schematic is used. With the jumper JP1 you can switch between 3.3V and 3.35V as VDD.



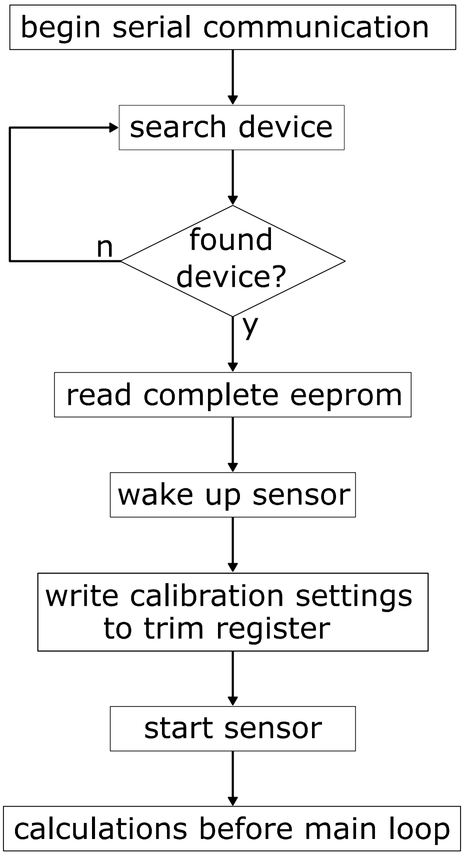
4 Sample Code

An Arduino program is defined in two parts: *setup* and *loop*. The *setup* function only runs once at the beginning. The *loop* function begins when *setup* is done. The following pages give an overview of the behavior of this functions.

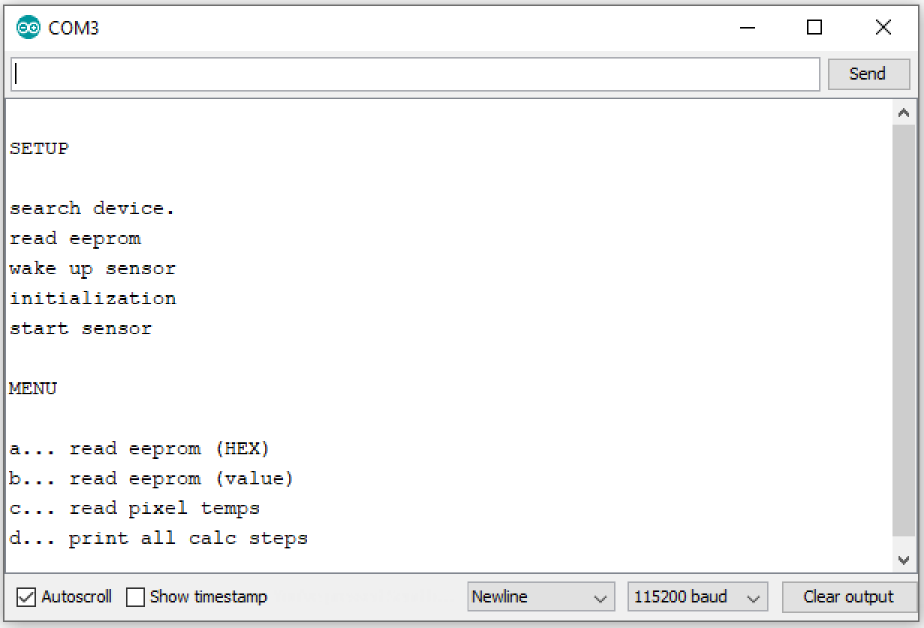
4.1 Serial mode

4.1.1 setup

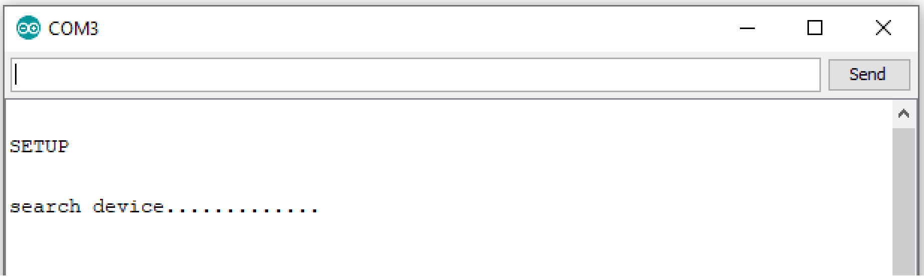
Here the steps during *setup* are shown:



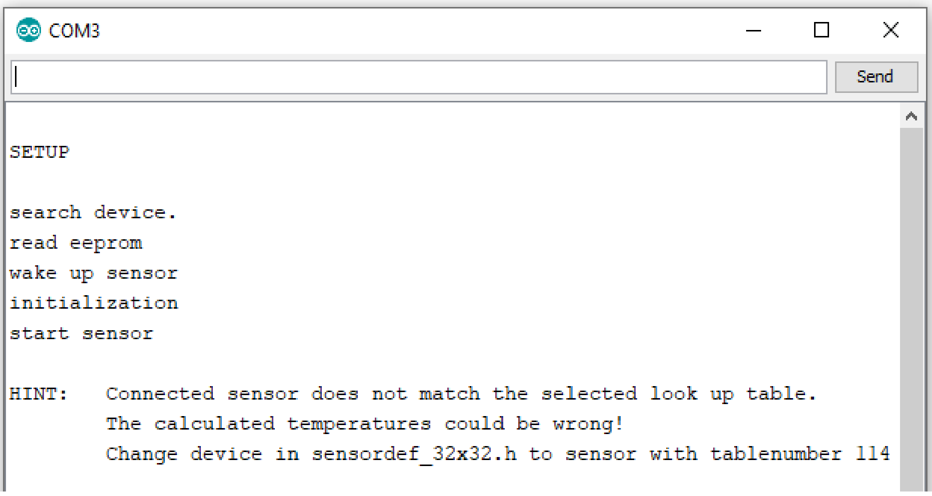
The serial monitor comments these setup phase:



If the setup is done the serial monitor shows the menu (part of the *loop* function). If there is no sensor connected, the setup rotates in an endless loop searching for a device (only for I2C devices).



You have to uncomment your sensor in sensordef32x32.h. A hint appears, if you choose the wrong sensor.

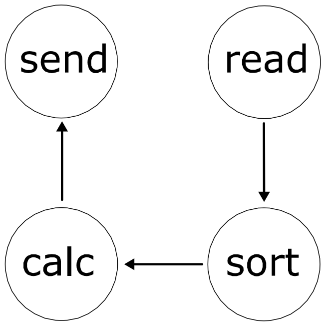


3.1.2 loop

When the menu is shown select one of the options:

a) print all values of the EEPROM in hexadecimal
b) print all important values of the EEPROM in their types (float, 8 unsigned int, …)
c) show current pixel temperatures in dK
d) show all calculation steps from raw data to final temperature picture

The first two points are only read out processes. The second two include the steps reading, sorting, calculating and sending (printing in Serial Monitor) of the pixel values. Order of the steps to get a temperature picture:



The **read** function (in sample code *read_pixel_data*) reads all blocks (first with PTAT and then with VDD) and the electrical offset. Between changing the configuration register and reading a block the function waits for the end of conversion bit (bit 0 in status register). Nothing happens during this time. The time between changing configuration register and reading block depends on clock frequency and ADC resolution.

Readout order (only fields with x are implemented):

	8x8	16x16	32x32	80x64	120x84
block 0 (with PTAT)	x	x	x	x	x
block 1 (with PTAT)		x	x	x	x
block 2 (with PTAT)			x	x	x
block 3 (with PTAT)			x	x	x
block 4 (with PTAT)					x
block 5 (with PTAT)					x
block 0 (with VDD)	x	x	x	x	x
block 1 (with VDD)		x	x	x	x
block 2 (with VDD)			x	x	x
block 3 (with VDD)			x	x	x
block 4 (with VDD)					x
block 5 (with VDD)					x
electrical offset	x	x	x	x	x

HINT:
The sample code of the Serial mode shows which values and steps are required to get a temperature picture. If you want to read continuously, use the faster state machine approach from Ethernet mode.

The **sort** function (in sample code *sort_data*) orders the blocks in an array. Sensors with more than 64 pixel have a readout order from outside to inside. Also here the averages of PTAT and VDD and the ambient temperature are calculated.

The **calc** function (in sample code *calculate_pixel_temps*) calculates the pixel temperature. The function includes the following steps:

1. Compensation of thermal offset
2. Compensation of the electrical offset
3. VDD compensation
4. Multiply sensitivity coefficient
5. Find correct reference temperatures in lookup table and do a bilinear interpolation

If the sensor has defect pixel, the function *pixel_masking* overwrites the defect pixel temperature with the average of the of all selected nearest neighbors (depends on the DeadPixMask of the pixel). The 8x8d sensor does not include VDD compensation and pixel_masking.

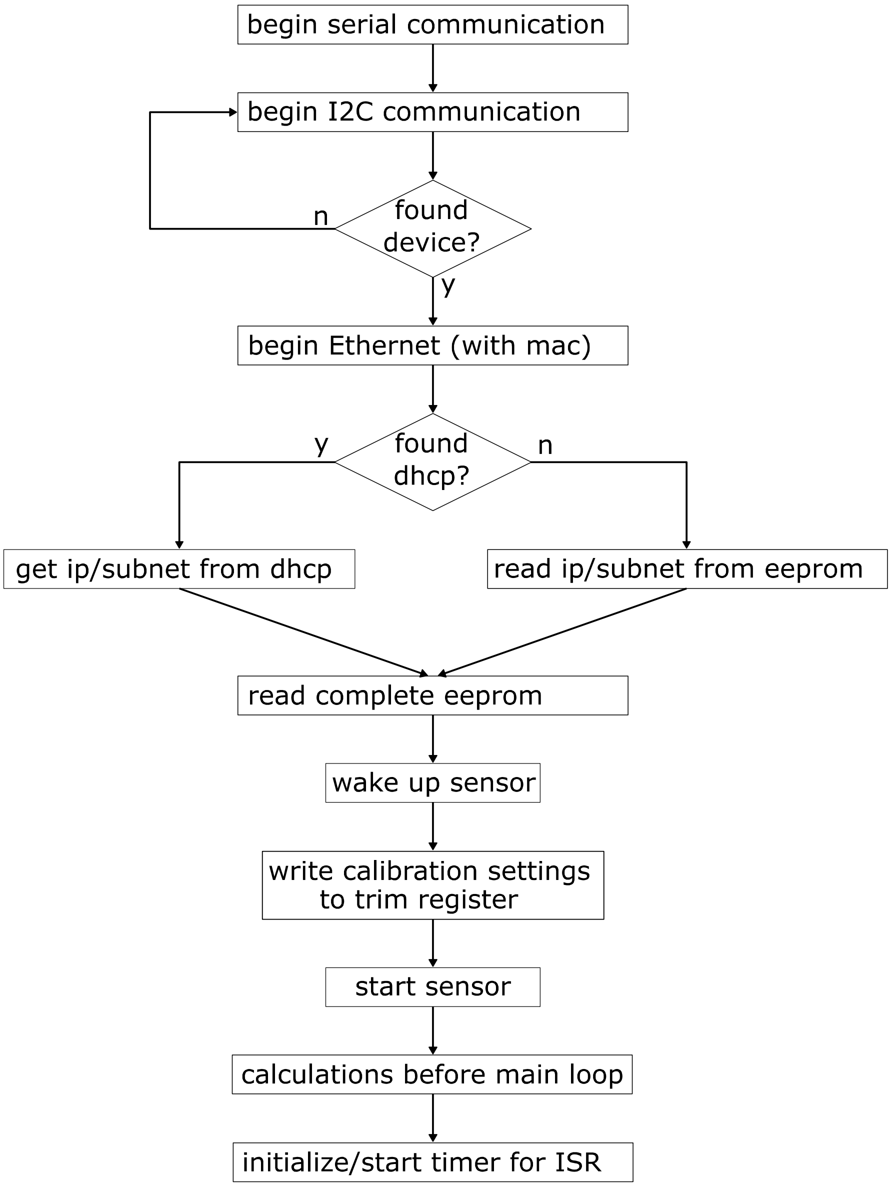
The **send** functions (in sample code *print_pixel_temps* and *print_calc_steps*) send the results to serial monitor. Printed steps (6. and 10. not for 8x8d):

1. row pixel voltages
2. electrical offset
3. ambient temperature (and PTAT average)
4. compensation thermal offset
5. compensation electrical offset
6. VDD compensation (and VDD average)
7. calculation sensitivity coefficient (calculated during setup)
8. multiply scaling coefficient and sensitivity coefficient to compensated voltages
9. pixel temperatures
10. pixel masking (only if the sensor has defect pixel)

3.2 Ethernet mode

3.2.1 setup

The steps during the setup are the same as in Serial mode. Only the selection of IP and subnet address and a timer interrupt are added. You have to change the MAC address in the sample code. You can usually find the MAC address on your Ethernet Shield.



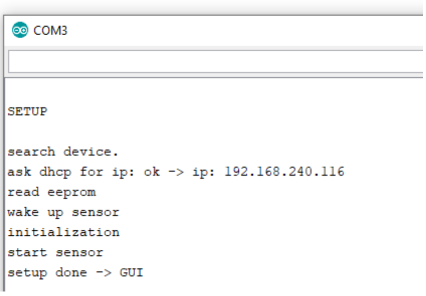
Hint: The Ethernet.h library defines a response time of 60 seconds. During this time the Arduino/Nucleo try to connect with a DCHP. If you are not using a DCHP and don't want to wait 60 seconds, change *timeout* and *responseTimeout* in Ethernet.h.

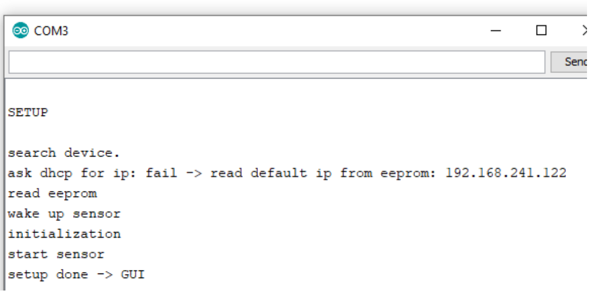
3.2.2 IP and subnet address

If the Arduino does not get an IP address from a DHCP, it uses the default address stored in the EEPROM. Register address for IP and subnet:

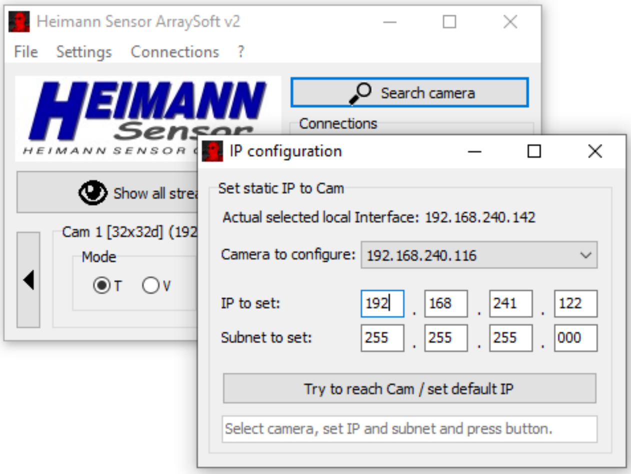
		EEPROM register			
example	*IP*	*192*	*168*	*241*	*122*
	Subnet	*255*	*255*	*255*	*0*
8x8	IP	0x05(MSB)	0x05(LSB)	0x06(MSB)	0x06(LSB)
	Subnet	0x3E(MSB)	0x3E(LSB)	0x3F(MSB)	0x3F(LSB)
16x16	IP	0x012(MSB)	0x012(LSB)	0x013(MSB)	0x013(LSB)
	Subnet	0x010(MSB)	0x010(LSB)	0x011(MSB)	0x011(LSB)
32x32	IP	0x021C	0x021D	0x021E	0x021F
	Subnet	0x0211	0x0212	0x0213	0x0214

The serial monitor shows the current IP.



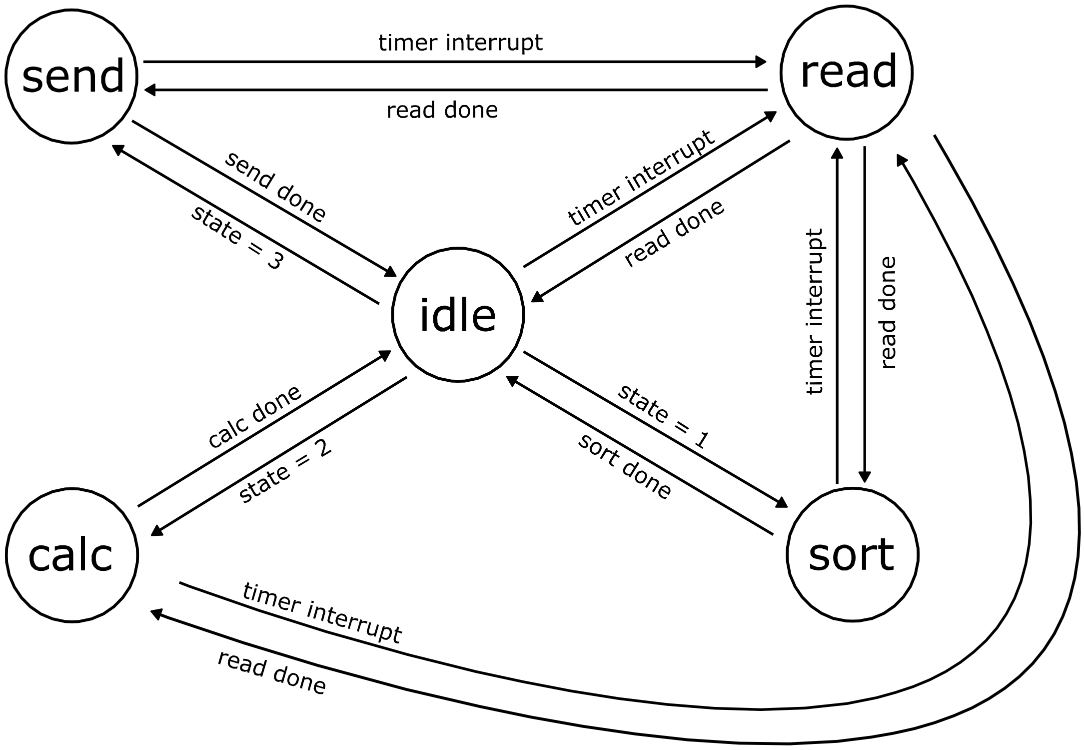


When the setup is done, the Arduino only communicates with the GUI. You can overwrite the IP/subnet in GUI settings: Heimann Sensor ArraySoft v2 -> Connections -> IP settings



3.2.3 loop

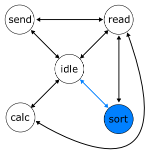
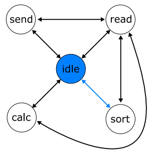
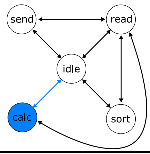
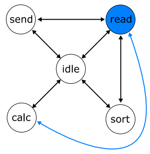
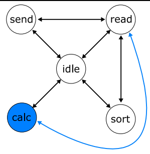
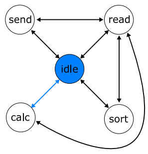
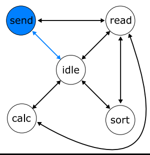
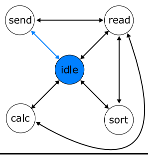
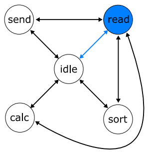
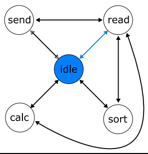
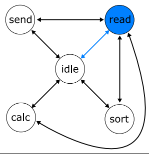
The behavior in the loop of the Ethernet mode can be described as a state machine.



The **sort** and **calc** function are the same as in Serial mode. The **read** function (in sample code *readblockinterrupt*) has the same role, but now the delays between changing the configuration register and getting a block data are used. A timer interrupts the current process and the program reads a block, saves it in global variable and changes the configuration register to the next block. PTAT and VDD alternates after each picture. Every tenth picture the program reads the electrical offset. The **send** function (in sample code *send_udp_packets*) communicates with the GUI, receives new commands and sends UDP packets with pixel temperatures or voltages. The **idle** function is symbolic for the *loop*, if nothing happens. Here the program only waits for next state or interrupt.

state	
0	- last picture was send and current picture is not complete
1	- sort current picture in an array - next picture can be read
2	- calculate pixel temperatures of current picture - next picture can be read
3	- send current picture to GUI - next picture can be read

example: 16x16d (next picture number to read = 10 | picture 9 (with VDD) is complete)

State	Mode		
1	**SORT**		- sort blocks of picture 9 in array form - calculate ambient temperature and average of PTAT and VDD - use PTAT from picture 8 - use new VDD from picture 9 - at the end: state = 2
2	**IDLE**		- check state
2	**CALC**		- calculate temperature of each pixel (picture 9) - at the end: state = 3
2	**READ**		timer interrupt! - read block 0 top and bottom of picture 10 (with PTAT) - set configuration register to block 1 (with PTAT) - at the end: back to last function
2	**CALC**		- calculate temperature of each pixel (picture 9) - at the end: state = 3
3	**IDLE**		- check state
3	**SEND**		- send udp packets - receive new commands from GUI - at the end: state = 0
0	**IDLE**		- check state - wait for state = 1
0	**READ**		timer interrupt! - read block 1 top and bottom of picture 10 (with PTAT) - set configuration register to electrical offset
0	**IDLE**		- check state - wait for state = 1
0	**READ**		timer interrupt! - read electrical offset top and bottom of picture 10 - set configuration register to block 0 (with VDD) - at the end: state = 1;
1	**SORT**	…	- sort blocks of picture 10 in array form …