Application note: greenhouse watering project

Contents

1 INTRODUCTION

1.1 Objective of this document

The objective of this document is to describe how to use WawiLib in a real automation example. The example is a program to control water distribution to a greenhouse, based on the moisture level of the soil.

The application is built as a real industrial automation application. It contains reading and scaling of an analog input value, a finite state machine (aka “Grafcet”) with automation logic and alarming. All parameters and controls can be modified using WawiLib.

The WawiLib “Getting started” demos use simple C language commands. This application takes us to another level: a framework for full-featured industrial automation applications. It shows how to implement multiple timers without blocking the main Arduino loop (not using delay()) and a safe and reliable way to implement sequential logic (finite state machine).

WawiLib will be used as a tool to test the application and as a tool to observe and operate the Arduino application. The demo uses USB for convenience but WiFi and Ethernet are also possible.

1.2 Greenhouse watering application functional description

The application uses a soil moisture sensor to read the moisture level of the soil. The moisture level is scaled into a range 0...100% into the variable *moistPctIst*. The scaling range can be modified using WawiLib to change *moistAnaMax* and *moistAnaMin. moistAna* contains the raw input value read directly from the analog input of the Arduino.

Watering can be enabled or disabled setting the variable *wateringEnable to 1.* If watering is enabled, *moistPctSoll* is used as setpoint for the moisture level. If the soil gets too dry, a cycle will start where the water valve will be opened and closed repeatedly.

Watering has to be done carefully. Therefore, even if the soil is too dry, the water will only be opened for a limited period of time. After that the water valve will be closed to give the water time to settle. If the moisture level is still not OK, the cycle will restart. The watering cycle can be controlled using the variables *valveTimeOpenSec* and *valveTimeHoldSec*.

If watering is enabled and the level of moisture does not reach the requested level after a certain period of time, an alarm output will signal the abnormal situation. The alarm state can be checked using the variable *dryAlarm*. The maximum time soil moisture can go below its setpoint before generating an alarm is determined by *moistureTimeAlarmSec.*

TotalValveTime.mact contains the total amount of time the water valve was open.

The control logic is implemented using a finite state machine. The state of the machine can be checked reading the value of *valveStateString*. The application contains various timers. WawiLib can be used to read the actual values of these timers using the variables *t1.m_act* and *t2.m_act*. A detailed description of how the logic is implemented can be found later in this document.

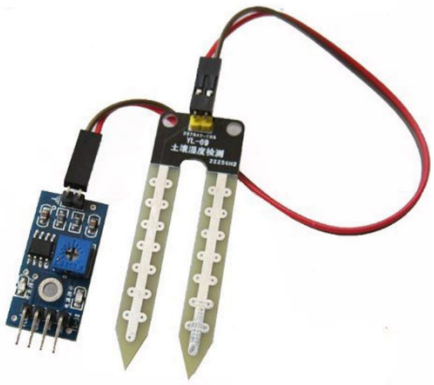
State changes of the FSM are reported in the output window so the user/developer can follow what is going on in a chronological order.

1.3 Software and hardware requirements

The Arduino IDE (in this example 1.8.12) and WawiLib both need to be installed on your PC. The demo runs with the licensed versions of WawiLib. With the unlicensed version you can use the demo concepts with 1 variable at a time.

Essential hardware you need is an Arduino (Mega) board, a USB programming cable, some Dupont male-male (breadboard) wires, a glass of water and a Windows PC (32 or 64 bit).

You also need a sensor that generates an analog signal 0-5V to determine the moisture level of the soil. In this example I use a sensor as in the picture below. It is an LM393 based converter. This type of sensor and conversion devices are available at many suppliers on the Internet. These signal amplifiers use an operational amplifier to create a voltage signal based on the resistance of the soil.



(If the hardware is not available you can use a potentiometer instead to simulate the output voltage of the sensor.)

The program uses I/O13 to control the valve (or a pump) and I/O12 to issue an alarm signal if moisture control is failing (soil is too dry for too long.)

The last thing you need is a glass of water to dip the sensor in to check if your application is working properly.

In this demo we will use the Arduino MEGA2560 board but other boards can be used in a similar or even identical way. (Be careful: some Arduino's are 3.3V and others are 5V based)

1.4 Required user experience

You should be familiar with the tutorials "Getting started with WawiLib USB" and "Debugging with WawiLib USB". There are no specific additional requirements. The C code used in this example

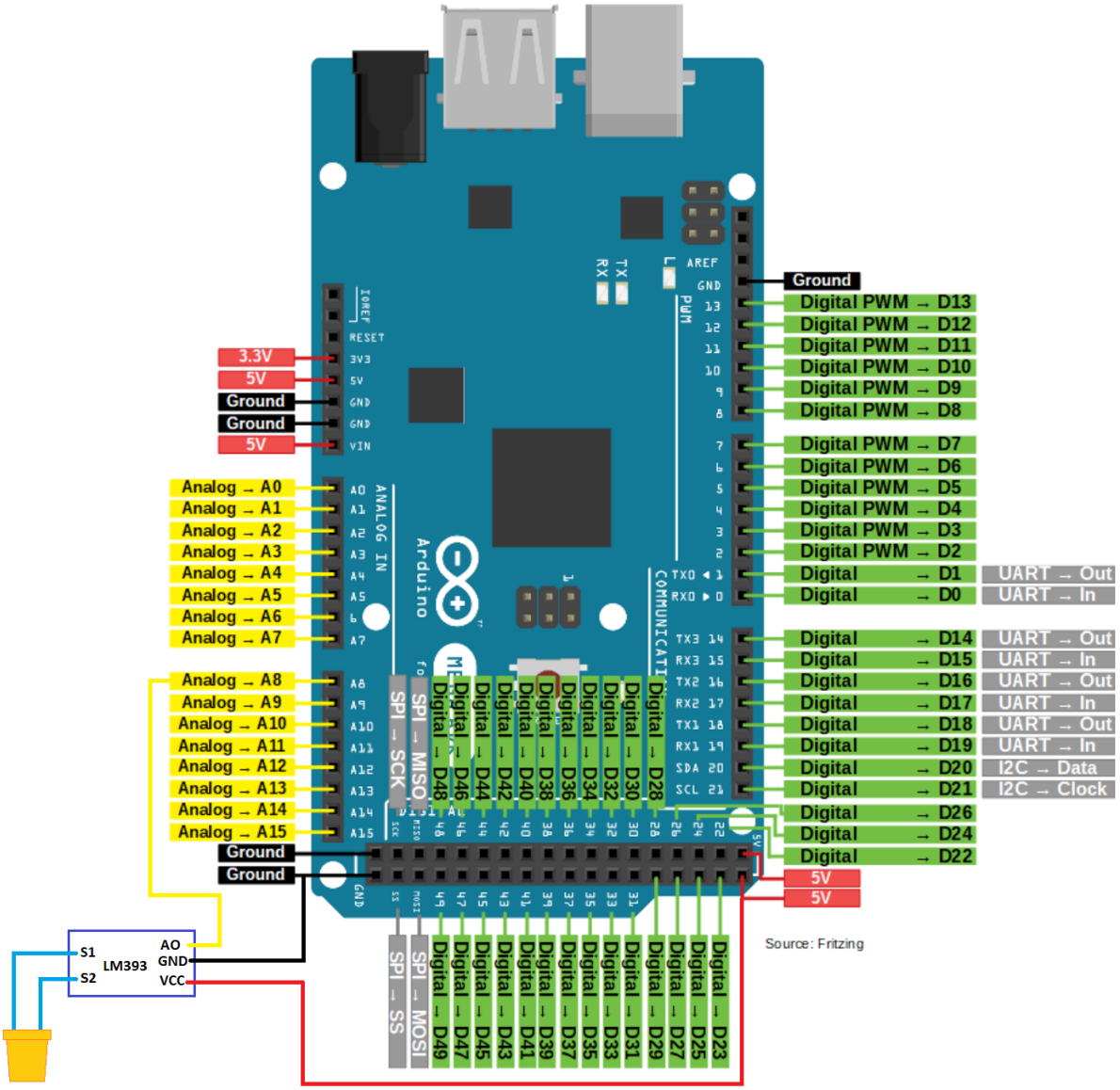
requires knowledge of the C language that is a bit more extended compared to the “Getting stared” examples.

The reason is that the concepts presented in this application can be used as a framework for other automation applications. I will first give a demo of the application as a whole and in the following chapters I will describe in detail the different parts of the application.

2 “WawiWaterSensorValve” live demo

2.1 Hardware connections

- ✓ Connect the GND pin of the interface converter to the GND of the Arduino (Pins next to pin 53).
- ✓ Connect the VCC pin of the interface converter to the +5V of the Arduino (Pins next to pin 22).
- ✓ Connect the moisture sensor to the 2 sensing pins on the interface converter (one of them has an “earth” symbol next to them).
- ✓ Connect the AO output signal of the interface converter to A8 of the Arduino (analog input 8).
- ✓ Connect the Arduino Mega to the PC using the USB cable.



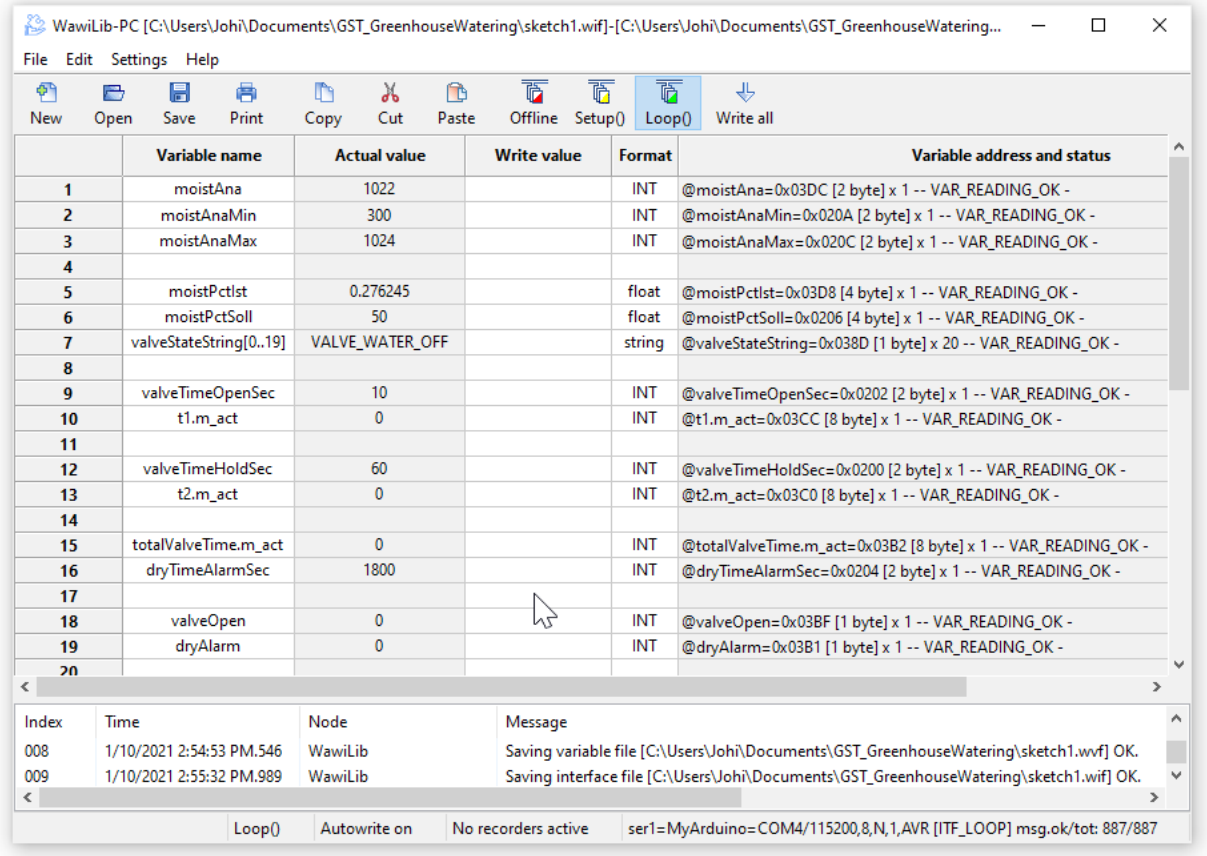
2.2 Load the demo

✓ Download the WawiWaterSensorValve.ino application from www.SylvesterSolutions.com. You can find the application in the download section of www.sylvestersolutions.com.

✓ Compile the demo and upload it to the Arduino board.

2.3 Visualize the variables of interest

✓ Start WawiLib and monitor the variables as indicated in the table below.

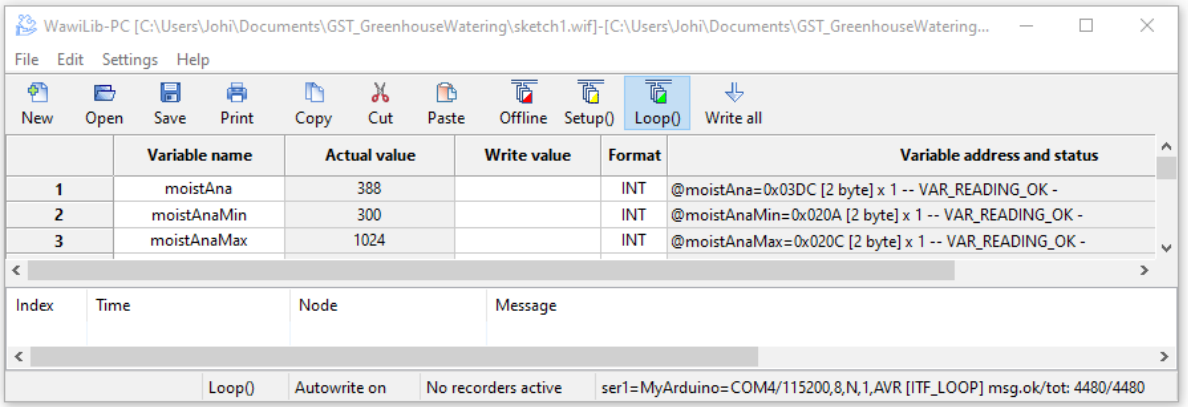


	Variable name	Actual value	Write value	Format	Variable address and status
1	moistAna	1022		INT	@moistAna=0x03DC [2 byte] x 1 -- VAR_READING_OK -
2	moistAnaMin	300		INT	@moistAnaMin=0x020A [2 byte] x 1 -- VAR_READING_OK -
3	moistAnaMax	1024		INT	@moistAnaMax=0x020C [2 byte] x 1 -- VAR_READING_OK -
4					
5	moistPctIst	0.276245		float	@moistPctIst=0x03D8 [4 byte] x 1 -- VAR_READING_OK -
6	moistPctSoll	50		float	@moistPctSoll=0x0206 [4 byte] x 1 -- VAR_READING_OK -
7	valveStateString[0..19]	VALVE_WATER_OFF		string	@valveStateString=0x038D [1 byte] x 20 -- VAR_READING_OK -
8					
9	valveTimeOpenSec	10		INT	@valveTimeOpenSec=0x0202 [2 byte] x 1 -- VAR_READING_OK -
10	t1.m_act	0		INT	@t1.m_act=0x03CC [8 byte] x 1 -- VAR_READING_OK -
11					
12	valveTimeHoldSec	60		INT	@valveTimeHoldSec=0x0200 [2 byte] x 1 -- VAR_READING_OK -
13	t2.m_act	0		INT	@t2.m_act=0x03C0 [8 byte] x 1 -- VAR_READING_OK -
14					
15	totalValveTime.m_act	0		INT	@totalValveTime.m_act=0x03B2 [8 byte] x 1 -- VAR_READING_OK -
16	dryTimeAlarmSec	1800		INT	@dryTimeAlarmSec=0x0204 [2 byte] x 1 -- VAR_READING_OK -
17					
18	valveOpen	0		INT	@valveOpen=0x03BF [1 byte] x 1 -- VAR_READING_OK -
19	dryAlarm	0		INT	@dryAlarm=0x03B1 [1 byte] x 1 -- VAR_READING_OK -
20					

Index	Time	Node	Message
008	1/10/2021 2:54:53 PM.546	WawiLib	Saving variable file [C:\Users\Johi\Documents\GST_GreenhouseWatering\sketch1.wvf] OK.
009	1/10/2021 2:55:32 PM.989	WawiLib	Saving interface file [C:\Users\Johi\Documents\GST_GreenhouseWatering\sketch1.wif] OK.

2.4 Reading and scaling the sensor input

✓ Put the sensor in a glass of water



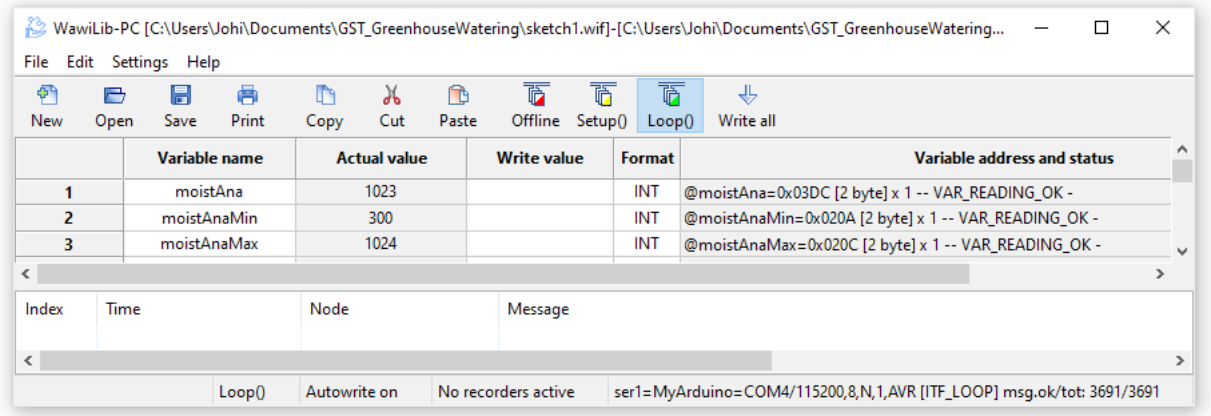
	Variable name	Actual value	Write value	Format	Variable address and status
1	moistAna	388		INT	@moistAna=0x03DC [2 byte] x 1 -- VAR_READING_OK -
2	moistAnaMin	300		INT	@moistAnaMin=0x020A [2 byte] x 1 -- VAR_READING_OK -
3	moistAnaMax	1024		INT	@moistAnaMax=0x020C [2 byte] x 1 -- VAR_READING_OK -

Index	Time	Node	Message

⇨ You should see the value of *moistAna* go down (= analog input 0..1023). Low resistance (= water between the electrodes) makes the value go down until 388 in my case. The output value of the signal converter is about 1.88V (Fluke multimeter measurement). As you go deeper with the sensor in the water, the value decreases.

⇨ The scaled value *moistPctIst* (moisture percentage calculated) will vary as the value of *moistAna* changes. But *moistPctIst* goes up as *moistAna* goes down. So *moistPctIst* increases as the soil becomes wetter. The idea is that moistPctIst is 100% for completely wet soil.

✓ Take the sensor out of the water (keep it in the air).

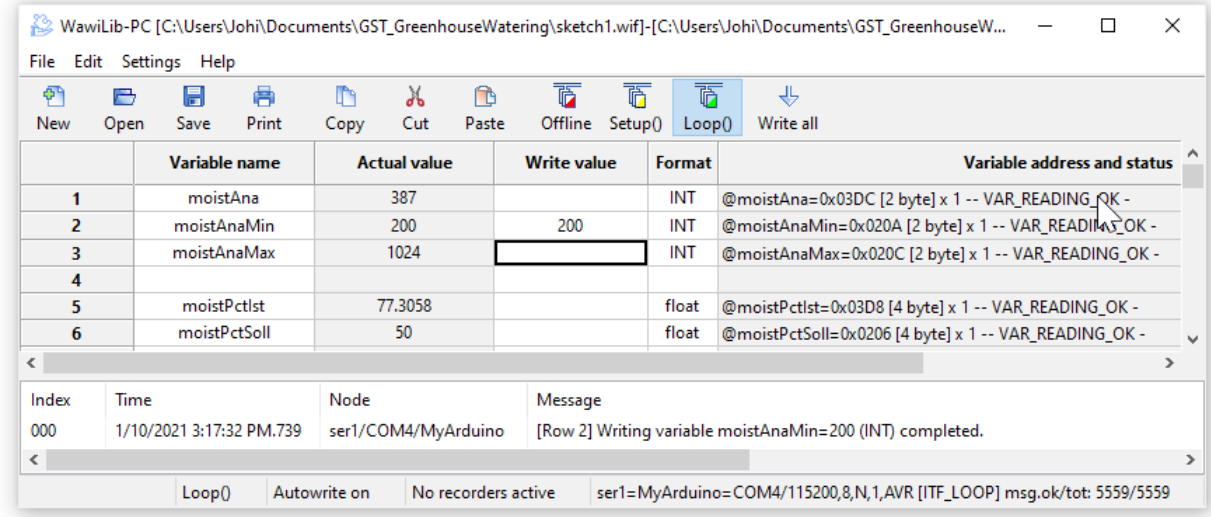


⇨ You should see the value of *moistAna* rise (= analog input 0..1024). High resistance (= no water) makes the value go up until 1023, the maximum value of the A/D on the Arduino board. In my case the output value of the signal converter is 4.96V (measured with my Fluke 117.)

⇨ Look at the value of *moistPctIst* = the actual moisture value, it goes down to practically 0%.

✓ Put the sensor back in the glass of water.

✓ Vary *moistAnaMin* and *moistAnaMax.* (Use WawiLib to change the value to these parameters.)

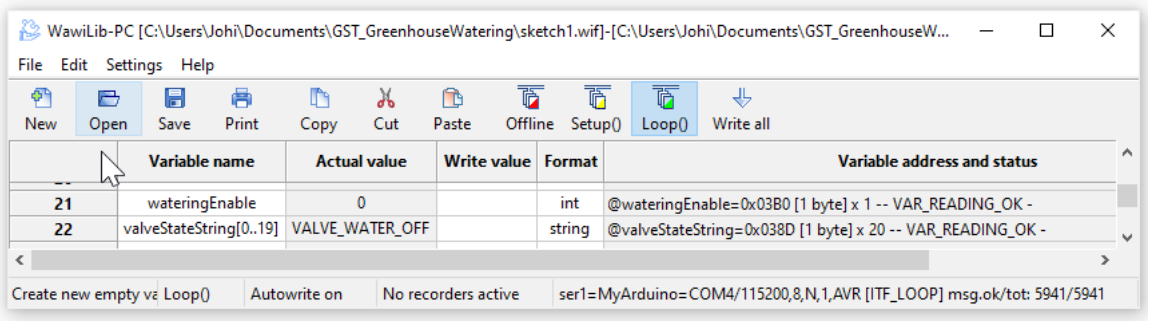


⇨ If you change the scaling factors *moistAnaMin* and *moistAnaMax,* you will see the calculated percentage of moisture change in line with the new scaling factors.

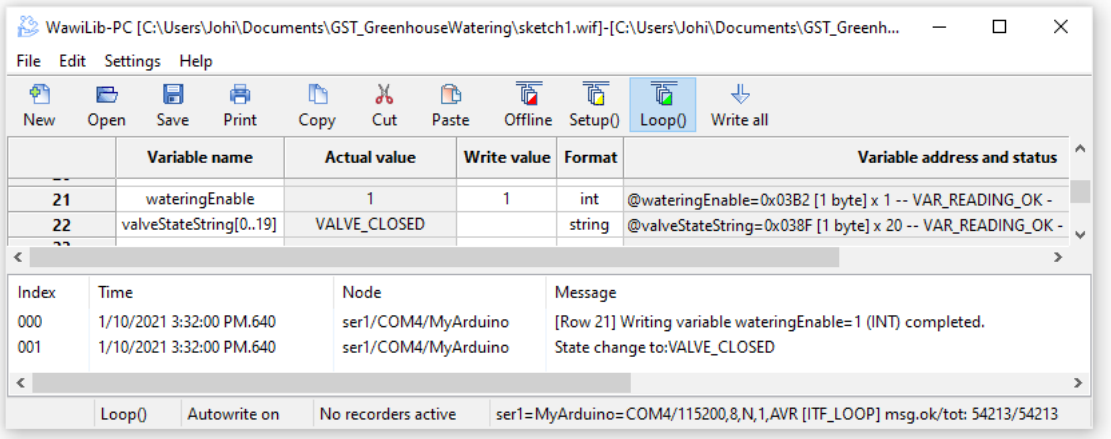
2.5 Activate moisture control (turn on watering)

✓ Look at the values of wateringEnable and *valveStateString.*

⇨ Watering is disabled and the FSM (finite state machine) is in the state *"VALVE_WATER_OFF"*.

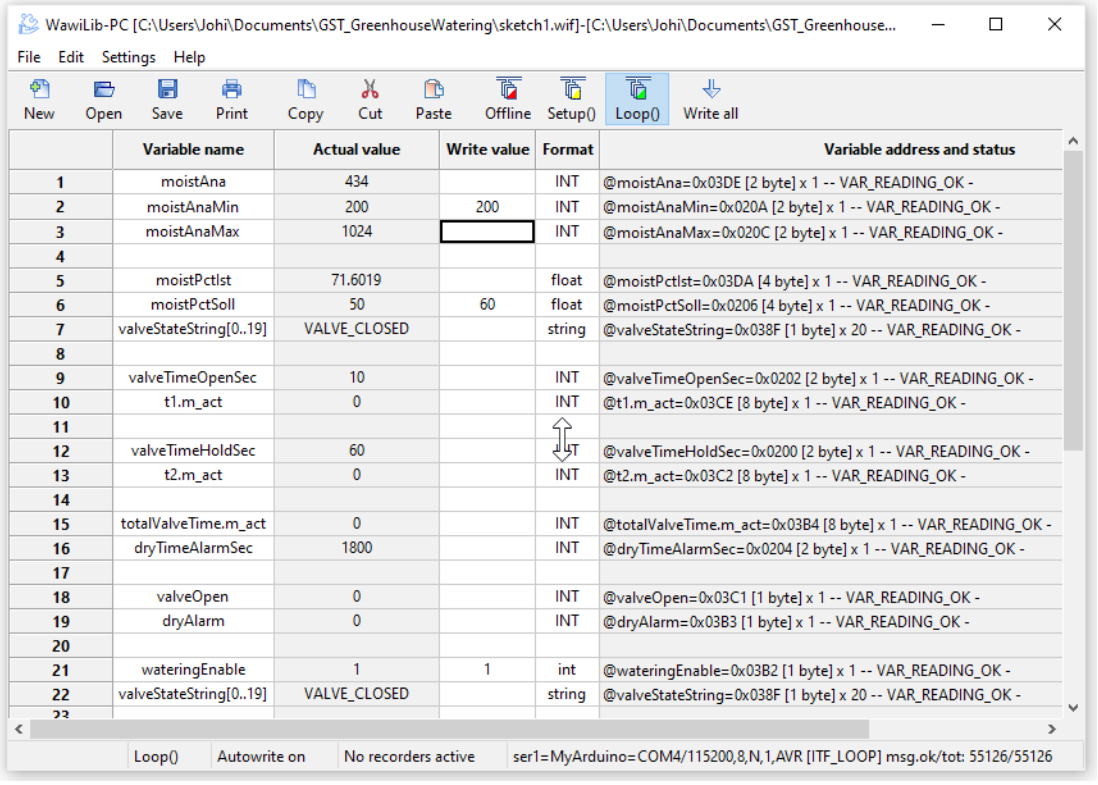


- ✓ Put the sensor in the water
- ✓ Make sure *moistPctSoll* is below *moistPctIst* (simulate soil wetter than necessary)
- ✓ Enable “display .print() messages” in the WawiLib output window.
- ✓ Write the value 1 to the variable *wateringEnable* via WawiLib.



⇨ As watering is enabled now, the FSM changes to the state *“VALVE_CLOSED”.*

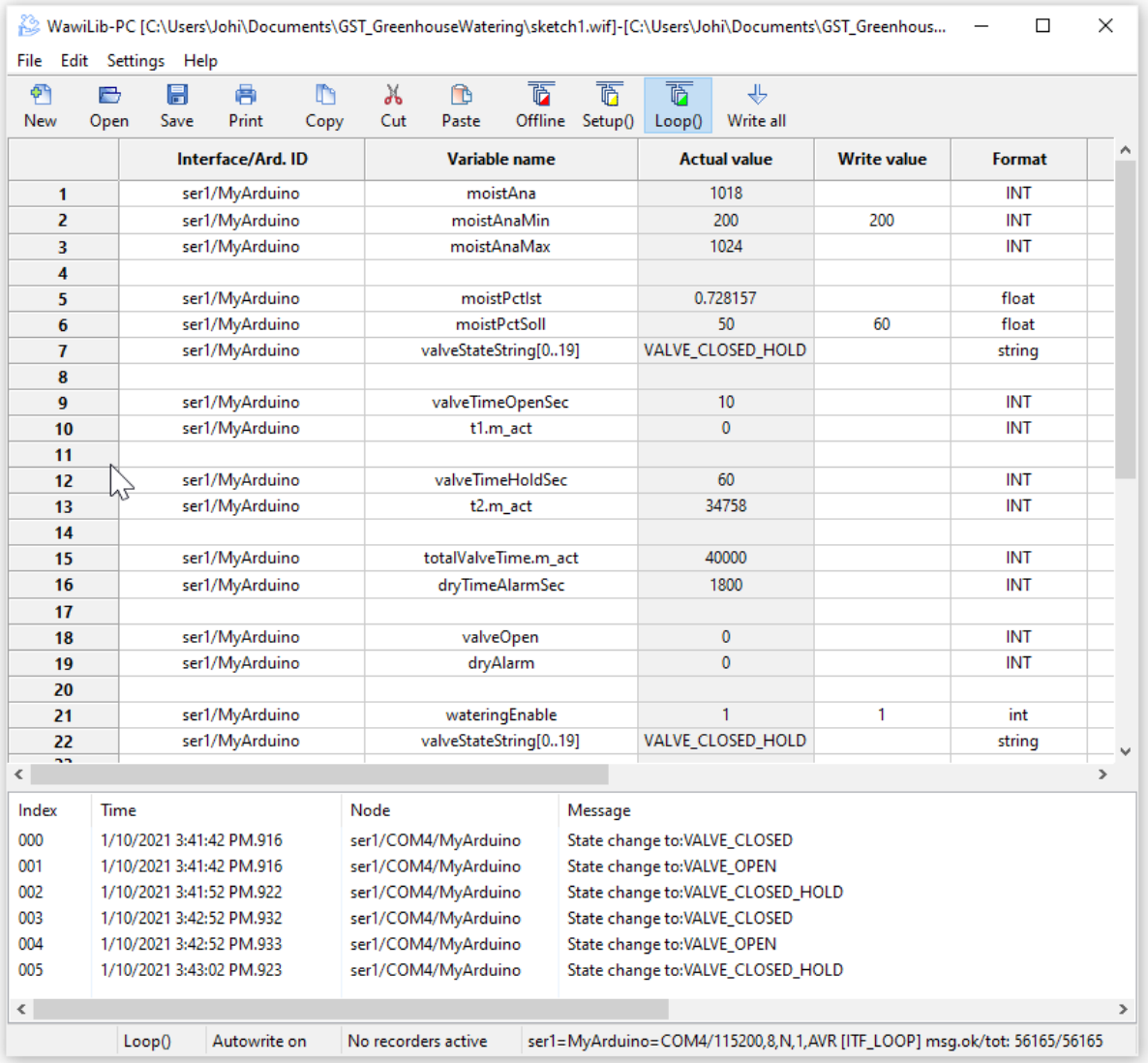
- ✓ Monitor the variables as indicated in the table below.



- ⇨ The state machine is in de state valve closed [line 9].
- ⇨ The valve is closed [22].
- ⇨ t1.m_act, the value of the time the valve is open, is 0 [12].
- ⇨ the value of *valveTimeOpenSec* (the setpoint of the time the valve is open) is 10 seconds [11].
- ⇨ t2.m_act, the value of the time to wait for the water to settle before restarting a cycle is 0 [15].
- ⇨ the value of *valveTimeHoldSec* (the setpoint of the settling time) is 60 seconds [14].
- ⇨ As watering is enabled now, the FSM changes to the state "VALVE_CLOSED" [9].

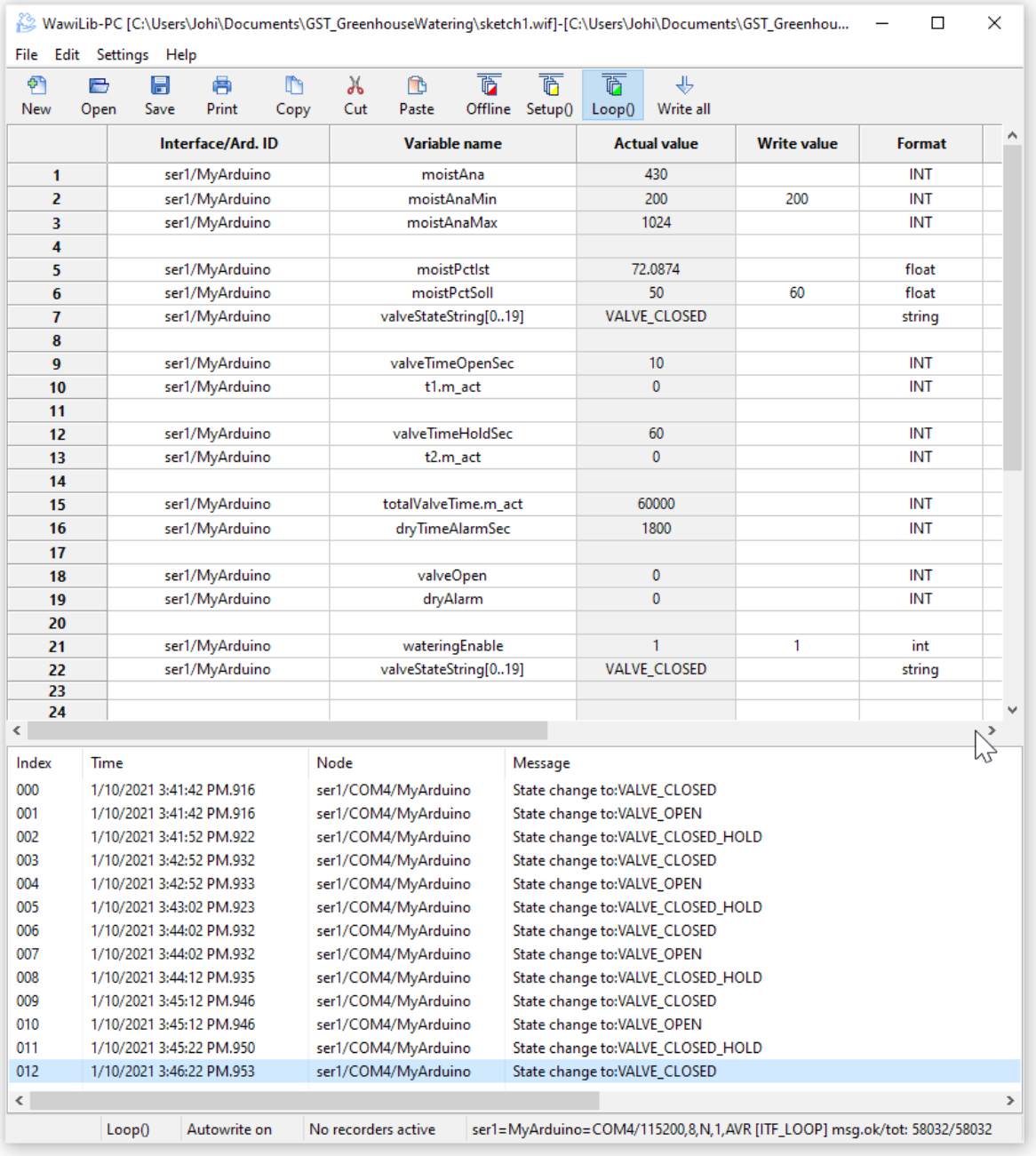
2.6 Simulate drought

- ✓ Remove the sensor from the water.



- ⇨ The value of *moistPctIst* changes to practically 0%.
- ⇨ The FSM, changes to the state *"VALVE_OPEN"* and *valveOpen* changes to 1.
- ⇨ The timer t1 decreases from 10000 (*valveTimeOpenSec* * 1000) to 0.
- ⇨ When t1 is 0 the FSM changes to *"VALVE_CLOSED_HOLD"* and *valveOpen* changes to 0.
- ⇨ The timer t2 decreases form 60000 *(valveTimeHoldSec* * 1000) to 0.
- ⇨ When *t2 is* 0 the FSM changes to *"VALVE_VALVE_OPEN."*
- ⇨ You can see the different cycles for watering, waiting for the result and again watering in the WawiLib-PC output window.

✓ Put the sensor back in the water.



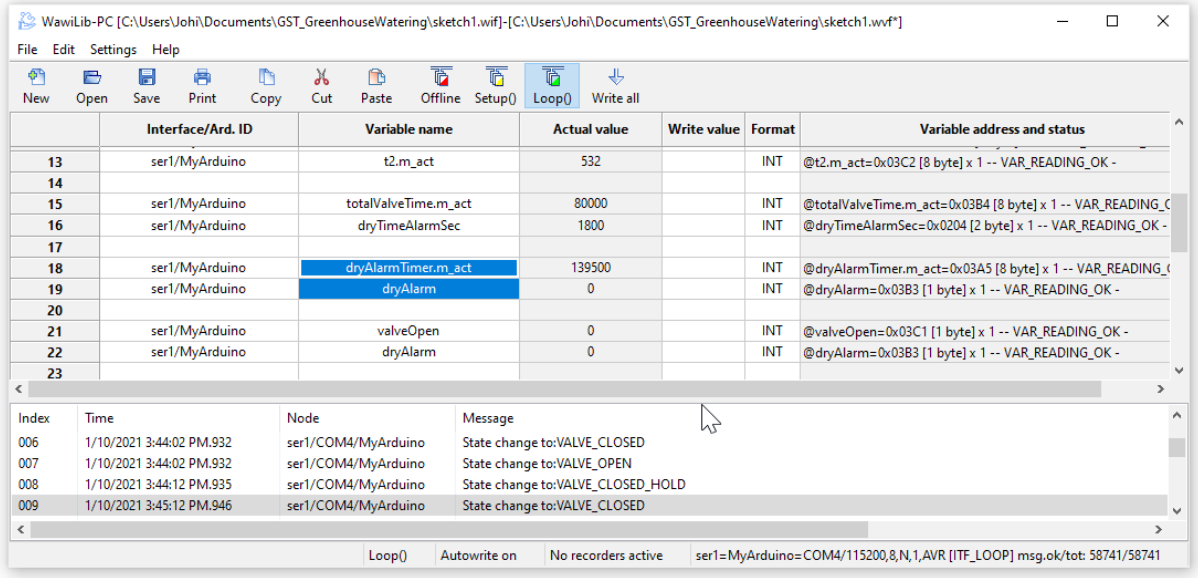
⇨ The cycle continues until t2 is 0 and then the FSM changes to *"VALVE_CLOSED"*

⇨ The variable "*valveOpen*" changes to 0 and stays there.

2.7 Alarm generation

✓ Remove the sensor from the water.

✓ Look at the variables as indicated in the table below.



⇨ The timer *dryAlarmTimer.m_act* increases.

⇨ As it reaches *dryTimeAlarmSec**1000, the status of *dryAlarm* goes high.

2.8 WawiLib as HMI

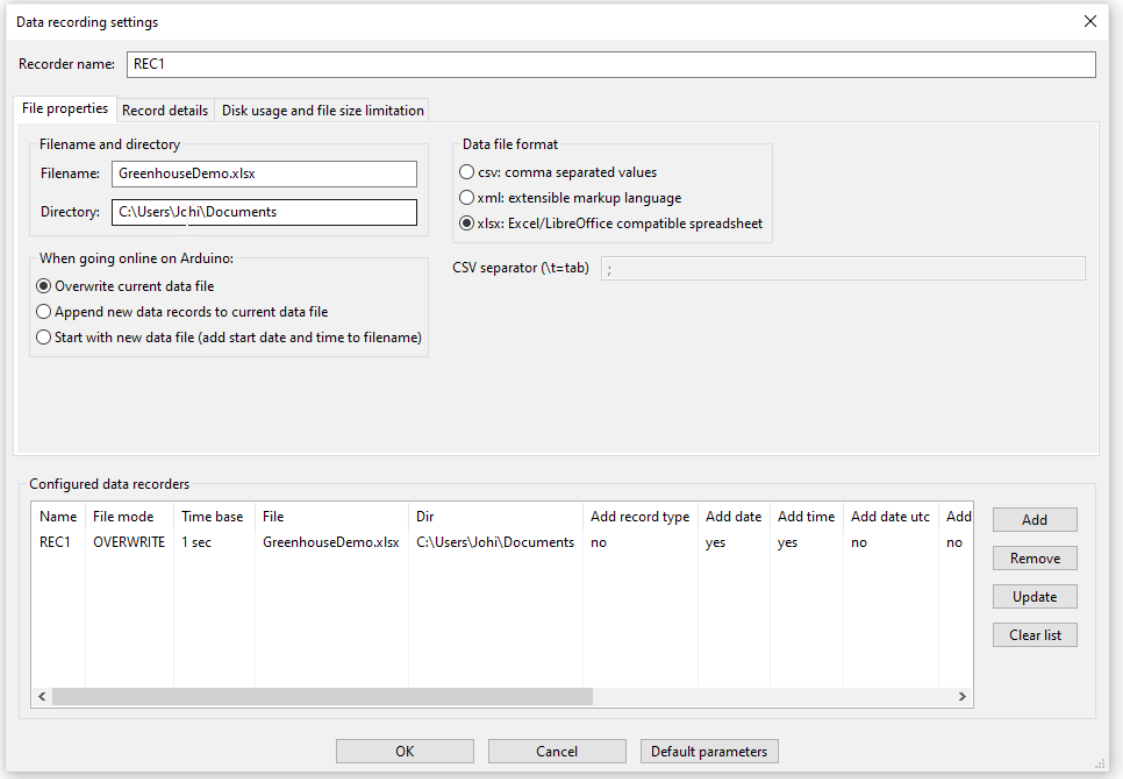
If you want to control the application remotely, you can use WawiLib to control the variable *wateringEnable* to turn watering on and off. The variable *moistPctSoll* can be used to control the setpoint and *moistPctIst* to observe the actual value. *TotalValveTime.m_act* contains the total time the water was running.

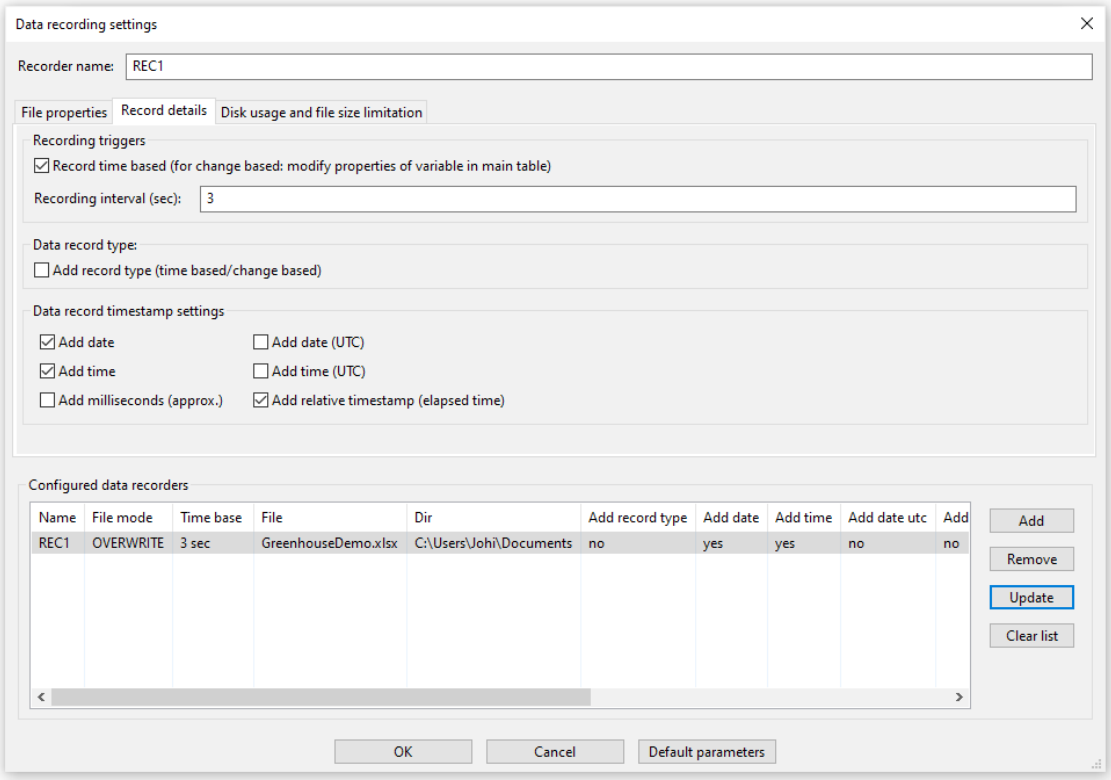
Observing and controlling the values of the variables explained in the previous paragraph gives you full control over your application. In the future, SylvesterSolutions will release a DLL that enables you to manipulate these variables using an application on the PC you have written yourself.

3 "WawiWaterSensorValve" data recording demo

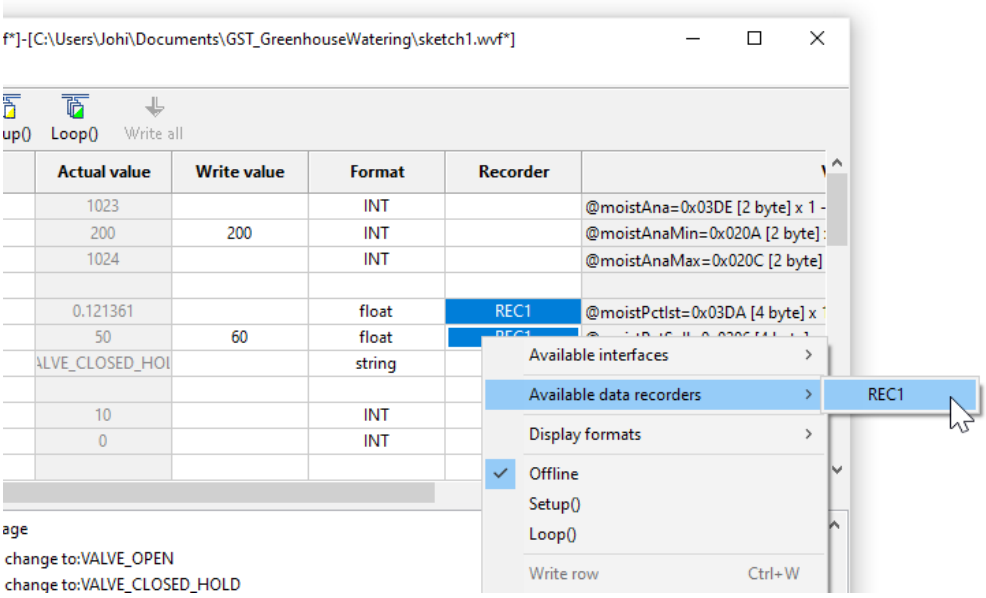
3.1 Continuous recording of the moisture level

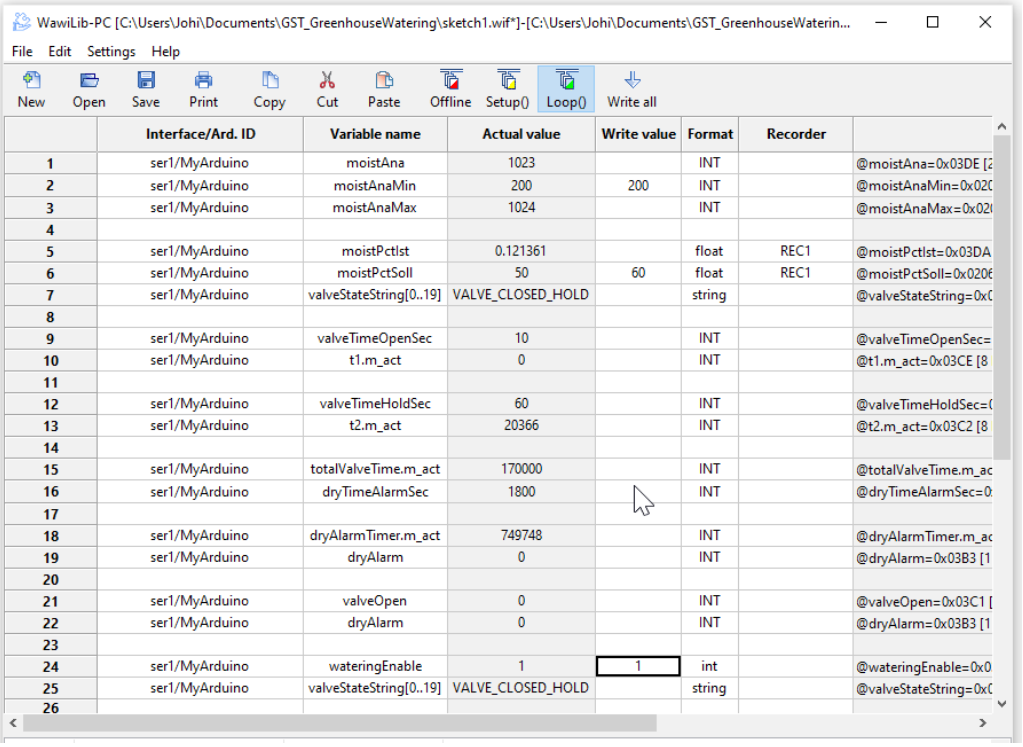
✓ Add a data recorder REC1 to *moistPctIst* and *moistPctSoll* (time-based 3 seconds for demo purposes):

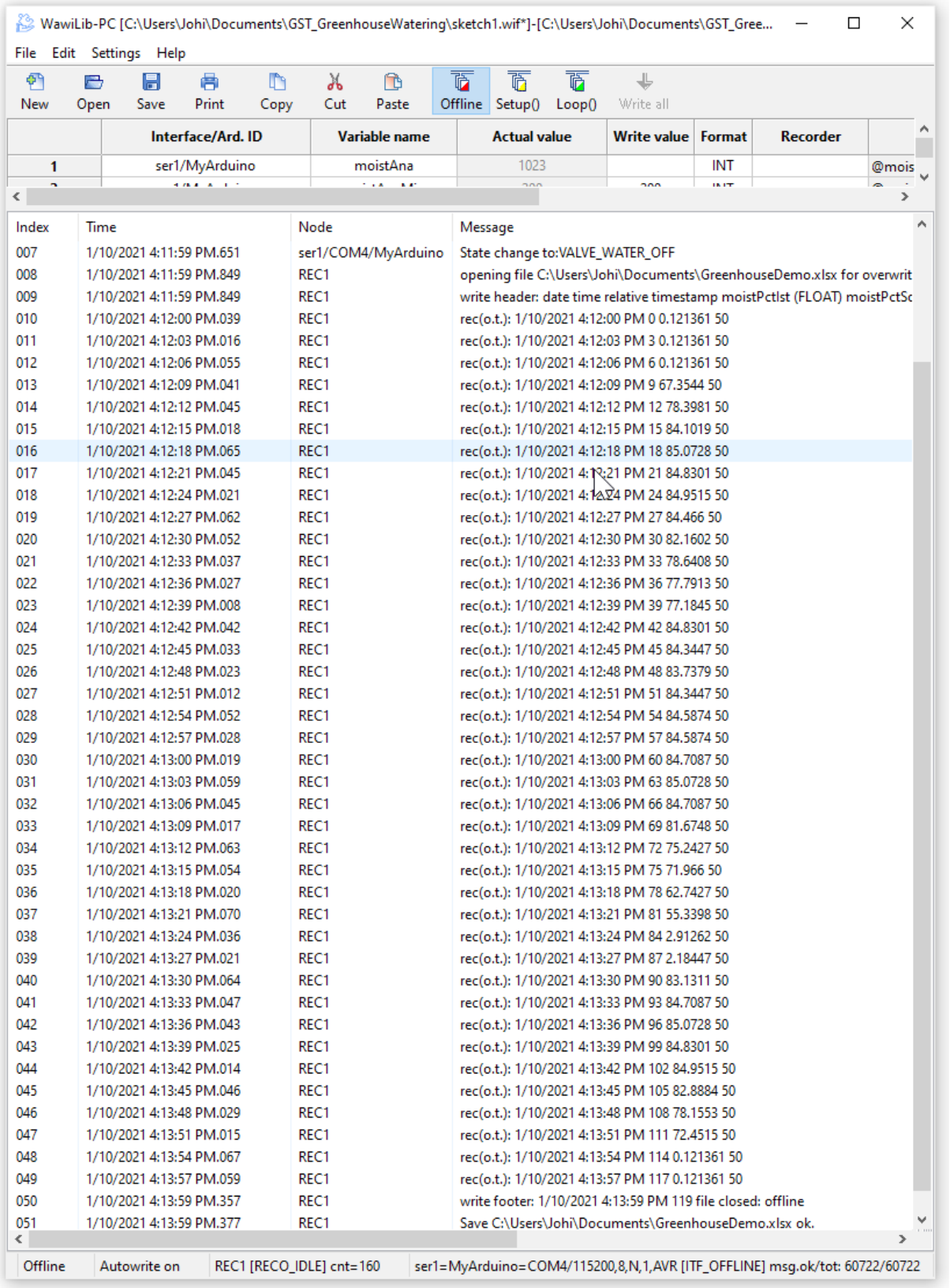




- ✓ Disable protocol tracing in the output window (click right on the output window).
- ✓ Disable .print() tracing in the output window.
- ✓ Enable recorder tracing in the output window.
- ✓ Slowly pull the sensor out of the water and put it back in.
- ✓ Repeat this action multiple times.
- ✓ Select the variables moistPctIst and moistPctSoll for recording (add text REC1 to the recorder column).

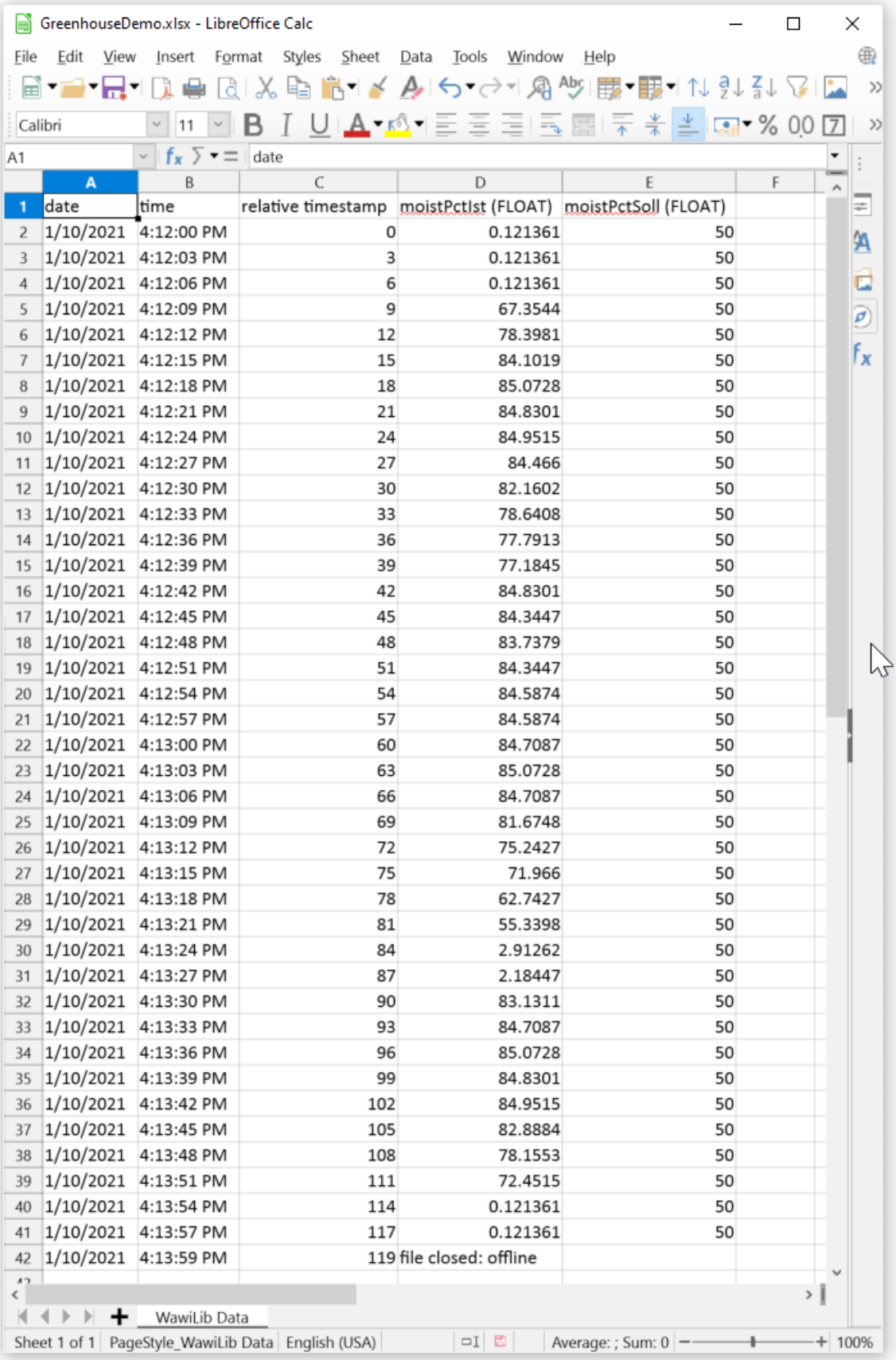






⇨ The values that are recorded change with the position of the sensor in the glass of water.

✓ Open the recorded .xlsx file in Microsoft Excel or LibreOffice calc:


GreenhouseDemo.xlsx - LibreOffice Calc

File Edit View Insert Format Styles Sheet Data Tools Window Help

Calibri 11

A1 date

	A	B	C	D	E	F
1	date	time	relative timestamp	moistPctIst (FLOAT)	moistPctSoll (FLOAT)	
2	1/10/2021	4:12:00 PM	0	0.121361	50	
3	1/10/2021	4:12:03 PM	3	0.121361	50	
4	1/10/2021	4:12:06 PM	6	0.121361	50	
5	1/10/2021	4:12:09 PM	9	67.3544	50	
6	1/10/2021	4:12:12 PM	12	78.3981	50	
7	1/10/2021	4:12:15 PM	15	84.1019	50	
8	1/10/2021	4:12:18 PM	18	85.0728	50	
9	1/10/2021	4:12:21 PM	21	84.8301	50	
10	1/10/2021	4:12:24 PM	24	84.9515	50	
11	1/10/2021	4:12:27 PM	27	84.466	50	
12	1/10/2021	4:12:30 PM	30	82.1602	50	
13	1/10/2021	4:12:33 PM	33	78.6408	50	
14	1/10/2021	4:12:36 PM	36	77.7913	50	
15	1/10/2021	4:12:39 PM	39	77.1845	50	
16	1/10/2021	4:12:42 PM	42	84.8301	50	
17	1/10/2021	4:12:45 PM	45	84.3447	50	
18	1/10/2021	4:12:48 PM	48	83.7379	50	
19	1/10/2021	4:12:51 PM	51	84.3447	50	
20	1/10/2021	4:12:54 PM	54	84.5874	50	
21	1/10/2021	4:12:57 PM	57	84.5874	50	
22	1/10/2021	4:13:00 PM	60	84.7087	50	
23	1/10/2021	4:13:03 PM	63	85.0728	50	
24	1/10/2021	4:13:06 PM	66	84.7087	50	
25	1/10/2021	4:13:09 PM	69	81.6748	50	
26	1/10/2021	4:13:12 PM	72	75.2427	50	
27	1/10/2021	4:13:15 PM	75	71.966	50	
28	1/10/2021	4:13:18 PM	78	62.7427	50	
29	1/10/2021	4:13:21 PM	81	55.3398	50	
30	1/10/2021	4:13:24 PM	84	2.91262	50	
31	1/10/2021	4:13:27 PM	87	2.18447	50	
32	1/10/2021	4:13:30 PM	90	83.1311	50	
33	1/10/2021	4:13:33 PM	93	84.7087	50	
34	1/10/2021	4:13:36 PM	96	85.0728	50	
35	1/10/2021	4:13:39 PM	99	84.8301	50	
36	1/10/2021	4:13:42 PM	102	84.9515	50	
37	1/10/2021	4:13:45 PM	105	82.8884	50	
38	1/10/2021	4:13:48 PM	108	78.1553	50	
39	1/10/2021	4:13:51 PM	111	72.4515	50	
40	1/10/2021	4:13:54 PM	114	0.121361	50	
41	1/10/2021	4:13:57 PM	117	0.121361	50	
42	1/10/2021	4:13:59 PM	119	file closed: offline		

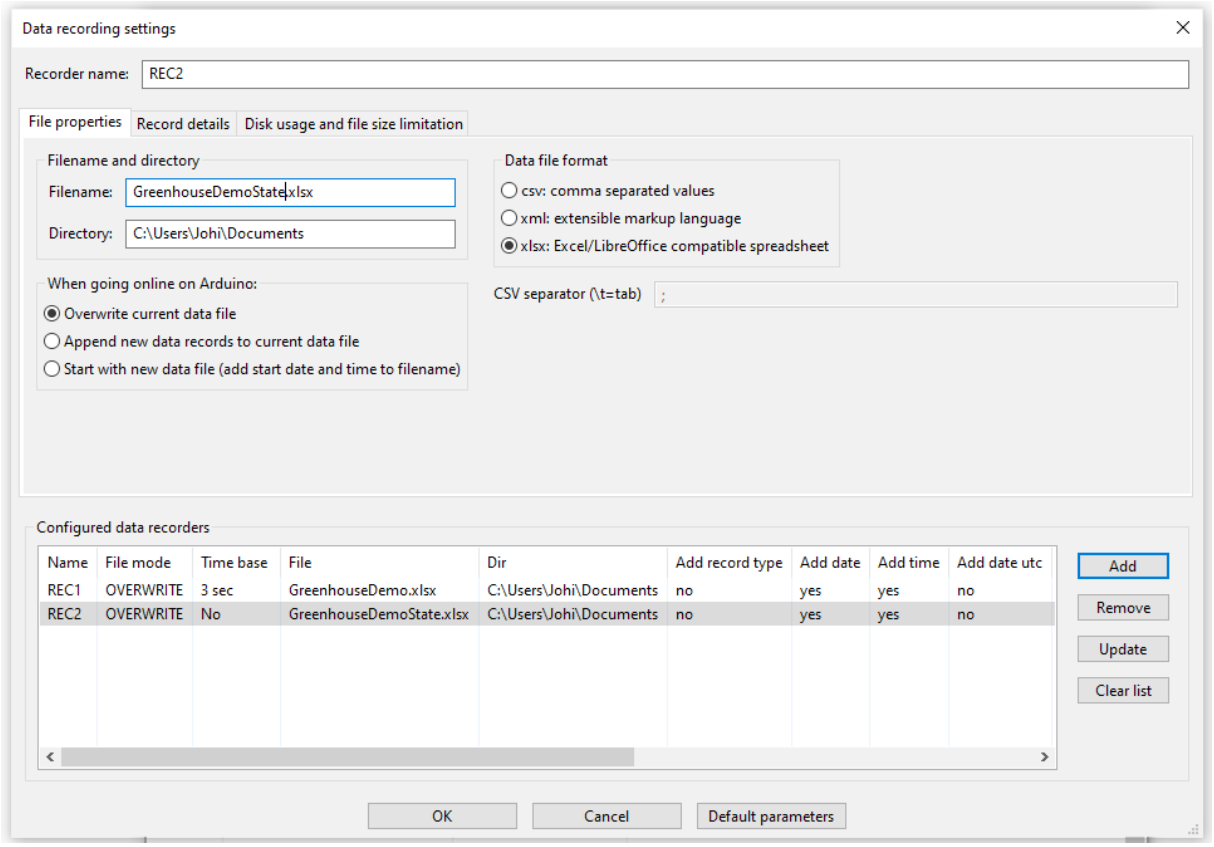
WawiLib Data

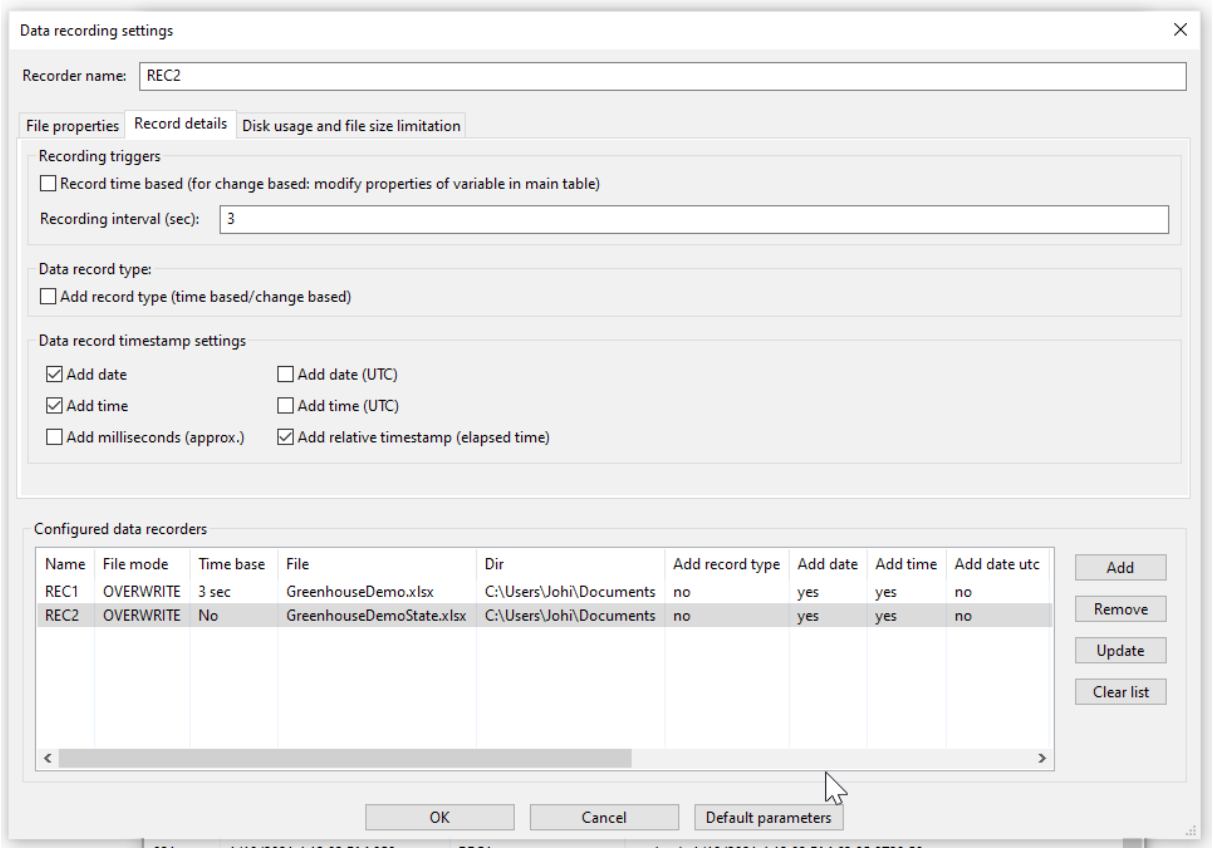
Sheet 1 of 1 PageStyle_WawiLib Data English (USA) Average: ; Sum: 0 100%

⇨ You can see the recorded values of the moisture setpoint and actual value.

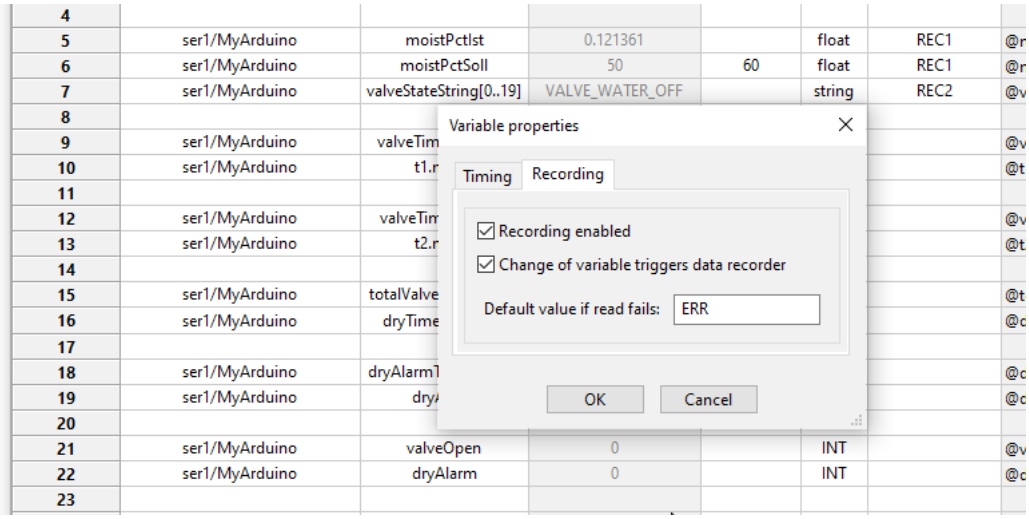
3.2 Change based recording of the state machine status.

✓ Add a data recorder to *valveStateString* and *valveOpen* (change based):



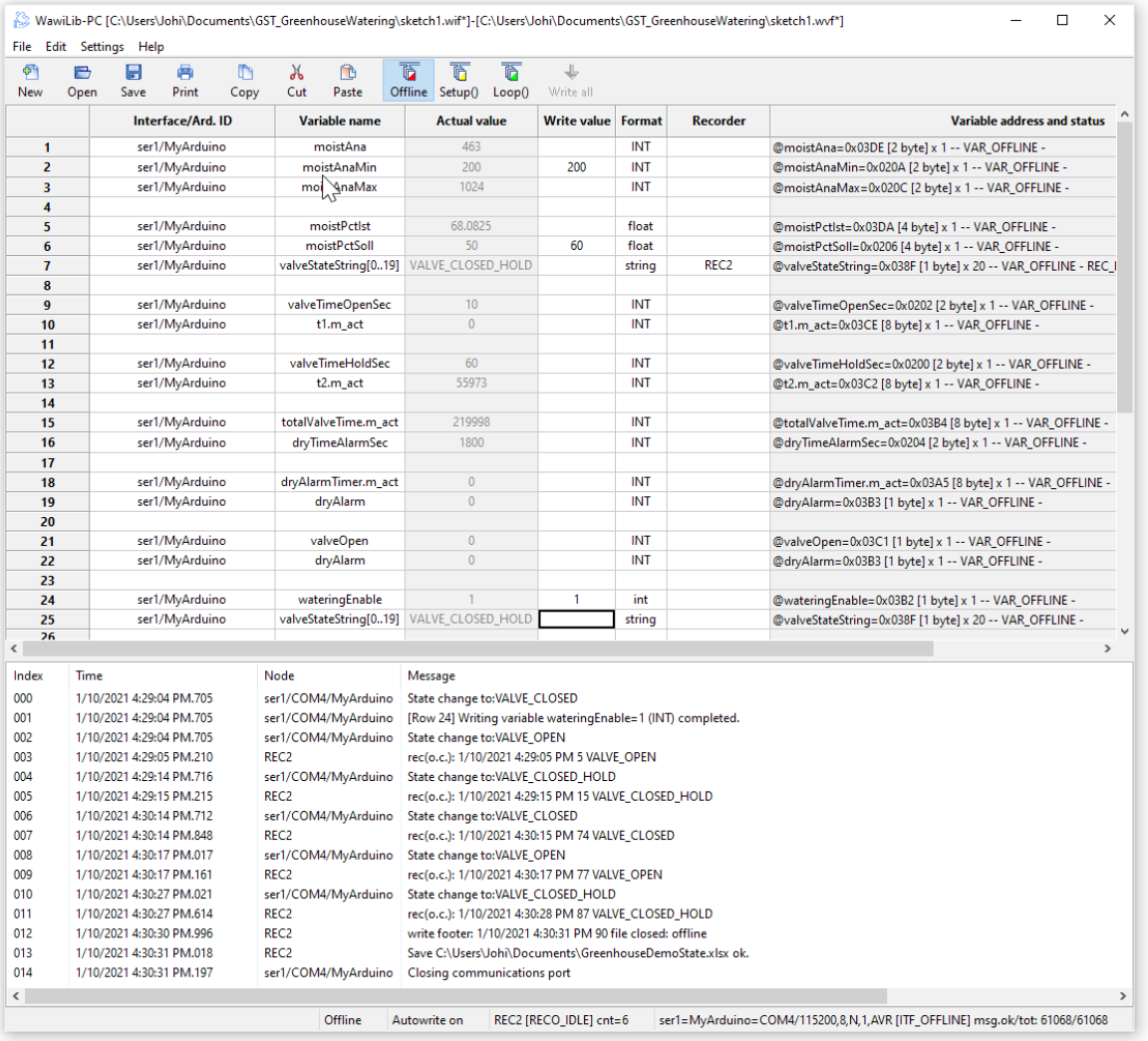


✓ Add REC2 as recorder for *valveStateString.*
✓ Change the properties of *valveStateString* as indicated below:



✓ Remove REC1 and at moistPctIst and moistPctSoll.
⇨ This in order not to clutter up the WawiLib-PC console output window. Recording with REC1 and REC2 at the same time is supported by WawiLib-PC.
✓ Go online, keep the sensor out of the water.

⇨ You can see the various state changes of the FSM recorded as they change. If the state does not change, there will be no record added to the recording file.

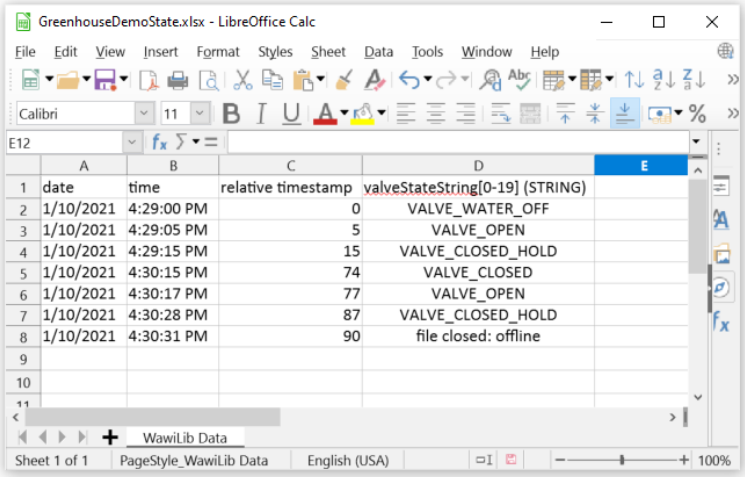


⇨ You see a message from WawiSerialUsb.print() each time the state of the FSM changes.

⇨ Once the PC reads this state change of the FSM, the data recorder writes the change to disk.

⇨ You see that data is only recorded if the state of the FSM changes, otherwise noting is recorded.

⇨ In the same way as described before you can open the recorded datafile in LibreOffice Calc or Excel:



	A	B	C	D	E
1	date	time	relative timestamp	valveStateString[0-19] (STRING)	
2	1/10/2021	4:29:00 PM	0	VALVE_WATER_OFF	
3	1/10/2021	4:29:05 PM	5	VALVE_OPEN	
4	1/10/2021	4:29:15 PM	15	VALVE_CLOSED_HOLD	
5	1/10/2021	4:30:15 PM	74	VALVE_CLOSED	
6	1/10/2021	4:30:17 PM	77	VALVE_OPEN	
7	1/10/2021	4:30:28 PM	87	VALVE_CLOSED_HOLD	
8	1/10/2021	4:30:31 PM	90	file closed: offline	
9					
10					

4 "WawiWaterSensorValve" Arduino software automation concepts

The objective of this part of the document is to document the main concepts used in the demo program. They come from the world of industrial automation.

4.1 Never put the main loop on hold

Essential in industrial automation applications is the main loop. This loop is run through many times per second. In this loop the logic is programmed to read the inputs, to calculate the result and to write the output signals.

For some applications, you would want to set an output signal OUT_A when an input signal IN_A comes high but after a time delay. In the same time, you would want to invert an input IN_B to an output OUT_B. The temptation is to use the delay instruction like this:

```
#define IO_IN_A 1
#define IO_OUT_A 2

#define IO_IN_B 11
#define IO_OUT_B 12

void setup() {
    pinMode(IO_IN_A, INPUT);
    pinMode(IO_OUT_A, OUTPUT);
    pinMode(IO_IN_B, INPUT);
    pinMode(IO_OUT_B, OUTPUT);
}
void loop()
{
    if (digitalRead(IO_IN_A))
    {
        delay(1000);
        digitalWrite(IO_OUT_A, HIGH);
    }
    else
        digitalWrite(IO_OUT_A, LOW);

    digitalWrite(IO_OUT_B, !digitalRead(IO_IN_B));
}
```

The problem is that the code that inverts input B is put on hold by the delay(1000) statement.

What we need is some kind of timer logic that we can set and check and that runs in parallel or independently. One of the ways to create timer logic is to use a hardware timer based interrupt to decrement a variable that is holding a time value in an interrupt handler.

The hardware timer approach has its disadvantages: First of all, it will eat up (much) CPU time executing the interrupt handler. You also do not know at what line of the main loop the interrupt handler will kick in. Therefore you cannot assume that through the main loop, the timer will have the same value.

Last but not least the variable holding the time value needs to be declared "volatile". This prevents compiler optimizations that make code fail when variables are shared between interrupt handlers and main code. The volatile keyword tells the compiler to always use the same memory location for the variable instead of using registers at some places and memory at other places for efficiency reasons.

4.2 The timer concept

Arduinos have a function called *millis().* This instruction returns the number of milliseconds since the board booted. In WawiWaterSensorValve.ino, each time the loop() passes, there is call to *timerLoopUp(UP_COUNT_TIMER &t)* or *timerLoopDown(DOWN_COUNT_TIMER &t)*. These 2 functions calculate the difference between the current and the previous value of *millis()* and increment or decrement the appropriate timer value in the same line. The symbol & means a call by reference so the original variable is modified and not a local copy.

```
//------------------------------------------------------------------------
// down count timer functions:
void timerLoopDown(DOWN_COUNT_TIMER& t)
{
    unsigned long m, delta;
    if (t.m_act > 0)
    {
        m = millis();
        delta = m - t.m_msPrev;
        t.m_msPrev = m;
        t.m_act = t.m_act - delta;
    }
    if (t.m_act < 0)
        t.m_act = 0;
}
```

```
//------------------------------------------------------------------------
// up count timer functions:
void timerLoopUp(UP_COUNT_TIMER& t, bool on)
{
    unsigned long m, delta;
    if (!t.m_onPrev && on)
        t.m_msPrev = millis();
    if (t.m_onPrev)
    {
        m = millis();
        delta = m - t.m_msPrev;
        t.m_msPrev = m;
        if (t.m_act < 0x7FFFFFFFFFFFFFFF - delta);
        t.m_act = t.m_act + delta;
    }
    t.m_onPrev = on;
}
```

The value of the time counter value and some status bits are kept together using a “C” structure to keep the code transparent. Since 32-bit time values can only go up to 4.294.967.295 ms (= 4.292.967 sec = 1.193 hr), we use 64-bit values for the timer so we are not confronted with overrun values.

```
//------------------------------------------------------------------------
// soft timer functions:
// these timers count using the millis() function of the arduino
struct DOWN_COUNT_TIMER
{
    long long m_act;    // actual value of the timer
    unsigned long m_msPrev; // helper value
};
```

```
struct UP_COUNT_TIMER
{
    long long m_act;    // actual value of the timer
    unsigned long m_msPrev; // helper value
    bool m_onPrev;  // helper value
};
```

The big advantage of this way of working is that timers are updated synchronously with the program in the main loop therefore they change value only at one well determined place and that the code is properly readable.

Timers are accessible using the functions *timerSet()* and *getValue()*. *TimerZero()* returns true if the timer is zero.

```
void timerSet(DOWN_COUNT_TIMER& t, int valueSec)
{
    t.m_act = (long long)valueSec * 1000;
    t.m_msPrev = millis();
}

long getValue(UP_COUNT_TIMER& t)
{
    if (t.m_act < (long long)0x7FFFFFFF * 1000)
        return t.m_act / 1000;
    else
        return 0x7FFFFFFF;
}

bool timerZero(DOWN_COUNT_TIMER& t)
{
    return (t.m_act == 0);
}
DOWN_COUNT_TIMER t1 = { 0.0,0.0 }; // valve open timer
DOWN_COUNT_TIMER t2 = { 0.0,0.0 }; // wait for water to settle timer
```

4.3 The finite state machine concept

4.3.1 Introduction

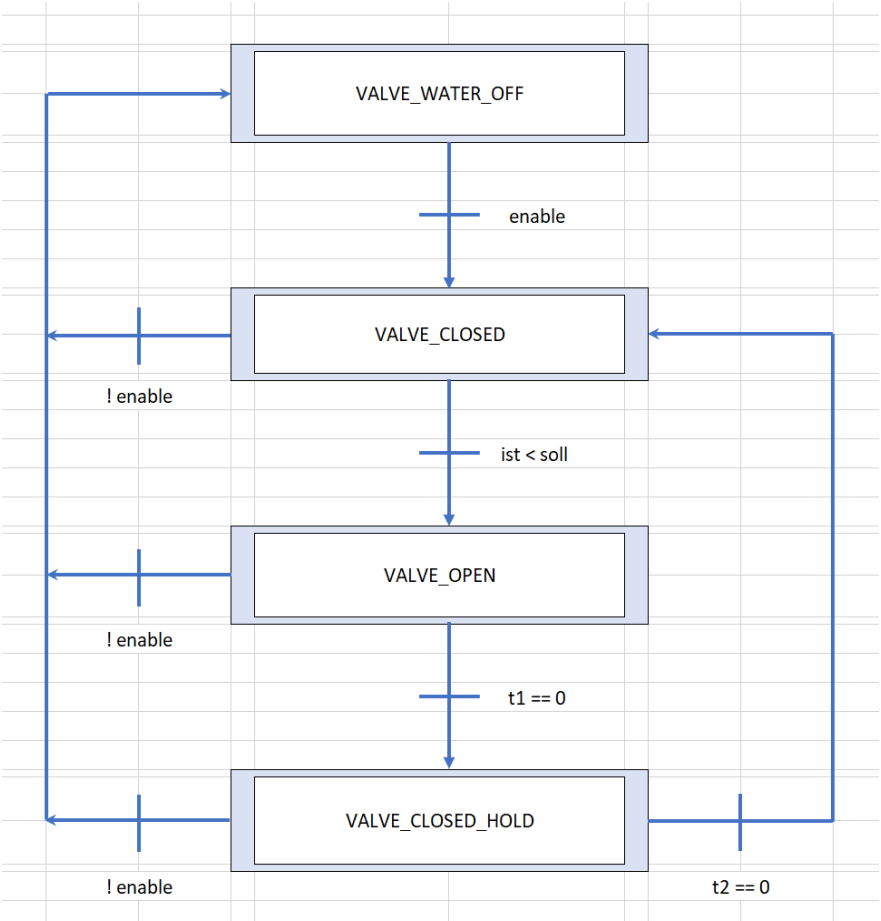
I have been writing software for more than 30 years now and I have been the witness of epic discussions of good and bad software. I have seen bad software work and good software fail.

Some time ago I read a small book from Mark Buelens, professor at the Vlerick Management school about decision making. The most important lesson he learned was: do focus on the quality of the decision process. The result of a decision can turn out good or bad because a lot of factors are involved, but whatever you do: focus on the quality of the process. My experience with software is similar, if you use methods that reduce the risk of failure you will get better results.

If you want to write a program that does some kind of sequencing, you can use a lot of Boolean logic to implement the program. But if you are not careful, you risk to create situations where your logic is blocked and hangs. Last minute modifications or small modifications create high risk.

4.3.2 Finite state machine concept

A finite state machine is a concept where the machine can only have a limited amount of states. It can only go from one defined state to another defined state using transitions. These transitions are well defined and only influence the transition from step A to B and nothing else.



This means if you want to implement additional transitions, you only modify a small part of the code that is isolated from other transitions. This makes the risk of destroying the code logic much smaller than in code that has been built without this type of logic.

In the figure above, you see the finite state machine for the WawiWaterSensorValve program. If the program is not active, the FSM is in the state *VALVE_WATER_OFF*.

In the state *VALVE_WATER_OFF*, the condition *enable* == true triggers the transition from *VALVE_WATER_OFF* to *VALVE_CLOSED*.

In the state *VALVE_VALVE_CLOSED,* the condition *ist < soll* (actual value of moisture measured is below the setpoint) triggers the transition from *VALVE_WATER_OFF* to *VALVE_OPEN*.

In the state *VALVE_VALVE_OPEN,* the condition t1==0 (count down timer has elapsed) triggers the transition from VALVE_OPEN to VALVE_CLOSED.

In the state *VALVE_WATER_OFF*, VALVE_OPEN and VALVE_CLOSED_HOLD the condition *enable* == false (!enable) triggers the transition from to *VALVE_WATER_OFF*.

Other transitions in the diagram work in the same way.

4.3.3 Finite state machine implementation

In C++ (11) programming, an enumeration type (also called enum) is a data type that consists of integral constants. This type of variable is ideal to represent the states of the state of the FSM.

```
//-----------------------------------------------------------------
// finite state machine variables to control the valve:
// (a finite state machine is a construct that can only have a limited number of
states)
enum class FsmState { VALVE_WATER_OFF = 0, VALVE_CLOSED = 1, VALVE_OPEN = 2,
VALVE_CLOSED_HOLD = 3 };
FsmState valveState, valveStatePrev{ FsmState::VALVE_WATER_OFF };
const char* stepStrings[] = { "VALVE_WATER_OFF", "VALVE_CLOSED", "VALVE_OPEN",
"VALVE_CLOSED_HOLD" };
char valveStateString[20];
```

If we want to display the actual value in WawiLib, we can use the variable *valveState* and visualize it as an integer number. This is in fact the poor man's solution as the numbers of the states do not give that much information.

Much better is to show the states of the FSM as text. To do this, I created an array of char* strings *stepStrings[]* and a variable *valveStateString* that will contain the actual value of the state machine in plain text.

StrCpy copies one of the strings in *stepStrings[]* into *valveStateString.* The appropriate array index is obtained by using the *valveState* as an integer.

```
    // report: translate valveState into a string and put it in valveStateString:
    strcpy(valveStateString, stepStrings[(int)valveState]);

    // Control I/O:
    if (valveState == FsmState::VALVE_OPEN)
        valveOpen = HIGH;
    else
        valveOpen = LOW;

    // display state change of the FSM in the output window:
    if (valveState != valveStatePrev)
    {
        valveStatePrev = valveState;
        WawiSrv.print("State change to: ");
        WawiSrv.println(valveStateString);
```

```
    }
```

The code below shows the implementation of the FSM itself. The case state using the enum *valveState* makes sure that the code remains transparent. The use of the timer concept explained in the previous chapter is used to implement timer functions.

```
//---------------------------------------------------------------------
// finite state machine:
void fsmValve()
{
    // Move from one step to another if conditions are right:
    switch (valveState)
    {
    case FsmState::VALVE_WATER_OFF:
        if (wateringEnable) valveState = FsmState::VALVE_CLOSED;
        break;
    case FsmState::VALVE_CLOSED:
        if (!wateringEnable) valveState = FsmState::VALVE_WATER_OFF;
        if (moistPctSoll > moistPctIst)
        {
            valveState = FsmState::VALVE_OPEN;
            timerSet(t1, valveTimeOpenSec);
        }
        break;
    case FsmState::VALVE_OPEN:
        if (timerZero(t1))
        {
            valveState = FsmState::VALVE_CLOSED_HOLD;
            timerSet(t2, valveTimeHoldSec);
        }
        if (!wateringEnable) valveState = FsmState::VALVE_WATER_OFF;
        break;
    case FsmState::VALVE_CLOSED_HOLD:
        if (timerZero(t2)) valveState = FsmState::VALVE_CLOSED;
        if (!wateringEnable) valveState = FsmState::VALVE_WATER_OFF;
        break;
                                                    }
```

4.3.4 Output control using finite state machines.

The last step is to set or clear the Arduino outputs based on the state of the FSM.

```
    // Control I/O:
    if (valveState == FsmState::VALVE_OPEN)
       valveOpen = HIGH;
    else
       valveOpen = LOW;
```

You can use different constructs to set outputs depending on the complexity of the program. Make sure that all outputs are set or cleared each time the code cycles though the main loop. Otherwise you risk to create parasite states in the FSM. This will be a cause of trouble sooner or later.

Do not clear all outputs by default and then set only the ones that need to be set depending on the state of the FSM. This will cause flicker on the outputs and can cause damage to the valve or pump motor.

4.3.5 Other parts

The completing part of the application is some code to implement an alarm that will be triggered if the moisture level is too low for a period of time. If you have been able to follow until now, the code

for the alarming will have no surprises. The code that drives WawiLib has been explained in the WawiLib “Getting Started” demos.

5 FURTHER READING

This demo demonstrates how to implement automation logic using a finite state machine and to test, debug and control the application using WawiLib. A very efficient concept of timers was introduced that does not block the main loop. A finite state machine (FSM) was used for analysis of the application. The FSM was translated in to Arduino C code using enums.

I hope you enjoyed this demo. Visit us on www.sylvestersolutions.com for the other demos.

6 APPENDIX: CODE LISTING

```
/*
* Project Name: WawiWaterSensorValve
* File: WawiWaterSensorValve.ino
* Description: demo file library for WawiSerialUsb libary.
* Reads water sensor, and contols water valve based on moisture level.
* Generates an alarm if moisture is below low level for too long
* Moisture level control can be turned on and off
* All parameters can be modified and observed using WawiLib
* Use programming USB port to make connection with the Arduino board.
* This software is provided as is without warranty of any kind
* Author: John Gijs.
* Created March 2020
* More info: www.sylvestersolutions.com
* Technical support: support@sylvestersolutions.com
* Additional info: info@sylvestersolutions.com
*/

#include <WawiSerialUsb.h>

#define VALVE_IO 13
#define DRY_ALARM_IO 13

//---------------------------------------------------------------------
// variables related to the actual moisture of the soil:
int moistAna = 0;          // analog input
int moistAnaMin = 300;     // scaling low limit
int moistAnaMax = 1024;    // scaling high limit

float moistPctIst = 0.0;   // actual value in %
float moistPctSoll = 50.0;  // low limit to start watering in %

//---------------------------------------------------------------------
// enable watering functionality is ON or OFF:
bool wateringEnable = false;
// actual state of watering valve itself and the alarm output:
bool valveOpen = false;
bool dryAlarm = false;

//---------------------------------------------------------------------
// alarm delay time:
int dryTimeAlarmSec = 1800;

//---------------------------------------------------------------------
// the water needs some time to settle before the moisture reaches the sensor
// so the valve needs to pulsate in order for the application to work properly:
int valveTimeOpenSec = 10;
int valveTimeHoldSec = 60;

//---------------------------------------------------------------------
// finite state machine variables to control the valve:
// (a finite state machine is a construct that can only have a limited number of
states)
enum class FsmState { VALVE_WATER_OFF = 0, VALVE_CLOSED = 1, VALVE_OPEN = 2,
VALVE_CLOSED_HOLD = 3 };
FsmState valveState, valveStatePrev{ FsmState::VALVE_WATER_OFF };
const char* stepStrings[] = { "VALVE_WATER_OFF", "VALVE_CLOSED", "VALVE_OPEN",
"VALVE_CLOSED_HOLD" };
char valveStateString[20];
```

```
//---------------------------------------------------------------------
// soft timer functions:
// these timers count unsing the millis() function of the arduino
struct DOWN_COUNT_TIMER
{
    long long m_act;    // actual value of the timer
    unsigned long m_msPrev; // helper value
};
struct UP_COUNT_TIMER
{
    long long m_act;    // actual value of the timer
    unsigned long m_msPrev; // helper value
    bool m_onPrev;  // helper value
};

//---------------------------------------------------------------------
// down count timer functions:
void timerLoopDown(DOWN_COUNT_TIMER& t)
{
    unsigned long m, delta;
    if (t.m_act > 0)
    {
        m = millis();
        delta = m - t.m_msPrev;
        t.m_msPrev = m;
        t.m_act = t.m_act - delta;
    }
    if (t.m_act < 0)
        t.m_act = 0;
}

void timerSet(DOWN_COUNT_TIMER& t, int valueSec)
{
    t.m_act = (long long)valueSec * 1000;
    t.m_msPrev = millis();
}

bool timerZero(DOWN_COUNT_TIMER& t)
{
    return (t.m_act == 0);
}
DOWN_COUNT_TIMER t1 = { 0.0,0.0 }; // valve open timer
DOWN_COUNT_TIMER t2 = { 0.0,0.0 }; // wait for water to settle timer

//---------------------------------------------------------------------------
// up count timer functions:
void timerLoopUp(UP_COUNT_TIMER& t, bool on)
{
    unsigned long m, delta;
    if (!t.m_onPrev && on)
        t.m_msPrev = millis();
    if (t.m_onPrev)
    {
        m = millis();
        delta = m - t.m_msPrev;
        t.m_msPrev = m;
        if (t.m_act < 0x7FFFFFFFFFFFFFFF - delta);
        t.m_act = t.m_act + delta;
    }
    t.m_onPrev = on;
```

```
}

void timerSet(UP_COUNT_TIMER& t, int valueSec)
{
    t.m_act = (long long)valueSec * 1000;
    t.m_msPrev = millis();
}

long getValue(UP_COUNT_TIMER& t)
{
    if (t.m_act < (long long)0x7FFFFFFF * 1000)
        return t.m_act / 1000;
    else
        return 0x7FFFFFFF;
}

UP_COUNT_TIMER totalValveTime = { 0.0,0.0,false }; // valve open total timer
UP_COUNT_TIMER dryAlarmTimer = { 0.0,0.0,false }; // system dry alarm timer

//-----------------------------------------------------------------------
// the wawilib part:
WawiSerialUsb WawiSrv;

void wawiVarDef()
{
    WawiSrv.wawiVar(moistAna);
    WawiSrv.wawiVar(moistAnaMin);
    WawiSrv.wawiVar(moistAnaMax);
    WawiSrv.wawiVar(moistPctSoll);
    WawiSrv.wawiVar(moistPctIst);
    WawiSrv.wawiVar(wateringEnable);

    WawiSrv.wawiVar(valveTimeOpenSec);
    WawiSrv.wawiVar(valveTimeHoldSec);

    WawiSrv.wawiVar(valveOpen);
    WawiSrv.wawiVar(t1.m_act);
    WawiSrv.wawiVar(t2.m_act);
    WawiSrv.wawiVarArray(valveStateString);
    WawiSrv.wawiVar(totalValveTime.m_act);
    WawiSrv.wawiVar(dryAlarmTimer.m_act);

    WawiSrv.wawiVar(dryAlarm);
    WawiSrv.wawiVar(dryTimeAlarmSec);
}

//-----------------------------------------------------------------------
// scaling and limiting functions for analog I/O processing:
int limit(int v, int min, int max)
{
    if (v < min) return min;
    if (v > max) return max;
    return v;
}

float scale(int v, int vMin, int vMax, float lo, float hi)
{
    return lo + (float)(v - vMin) / (vMax - vMin) * (hi - lo);
}
```

```
//---------------------------------------------------------------------
// finite state machine:
void fsmValve()
{
    // Move from one step to another if conditions are right:
    switch (valveState)
    {
    case FsmState::VALVE_WATER_OFF:
        if (wateringEnable) valveState = FsmState::VALVE_CLOSED;
        break;
    case FsmState::VALVE_CLOSED:
        if (!wateringEnable) valveState = FsmState::VALVE_WATER_OFF;
        if (moistPctSoll > moistPctIst)
        {
            valveState = FsmState::VALVE_OPEN;
            timerSet(t1, valveTimeOpenSec);
        }
        break;
    case FsmState::VALVE_OPEN:
        if (timerZero(t1))
        {
            valveState = FsmState::VALVE_CLOSED_HOLD;
            timerSet(t2, valveTimeHoldSec);
        }
        if (!wateringEnable) valveState = FsmState::VALVE_WATER_OFF;
        break;
    case FsmState::VALVE_CLOSED_HOLD:
        if (timerZero(t2)) valveState = FsmState::VALVE_CLOSED;
        if (!wateringEnable) valveState = FsmState::VALVE_WATER_OFF;
        break;
    }

    // report: translate valveState into a string and put it in valveStateString:
    strcpy(valveStateString, stepStrings[(int)valveState]);

    // Control I/O:
    if (valveState == FsmState::VALVE_OPEN)
        valveOpen = HIGH;
    else
        valveOpen = LOW;

    // display state change of the FSM in the output window:
    if (valveState != valveStatePrev)
    {
        valveStatePrev = valveState;
        WawiSrv.print("State change to: ");
        WawiSrv.println(valveStateString);
    }
}

//---------------------------------------------------------------------
// Make alarm if dry is too long high:
void alarming()
{
    // if watering is enabled, and moisture is too low a timer is started
    bool dry = moistPctIst < moistPctSoll;
    if (wateringEnable && dry)
        timerLoopUp(dryAlarmTimer, dry);
    else
        timerSet(dryAlarmTimer, 0);
```

```
    // alarm after moistureTimeAlarmSec seconds of drought
    if (getValue(dryAlarmTimer) > dryTimeAlarmSec)
        dryAlarm = HIGH;
    else
        dryAlarm = LOW;
}
//---------------------------------------------------------------------
void setup()
{
    Serial.begin(115200);
    WawiSrv.begin(wawiVarDef, Serial, "MyArduino");
    pinMode(VALVE_IO, OUTPUT);
    pinMode(DRY_ALARM_IO, OUTPUT);
}

void loop()
{
    // read and process moisture level:
    moistAna = limit(analogRead(8), 0, 1024);
    moistPctIst = scale(moistAna, moistAnaMin, moistAnaMax, 100.0, 0.0);
    // timers for valve open and valve open hold/delay:
    timerLoopDown(t1);
    timerLoopDown(t2);
    timerLoopUp(totalValveTime, valveOpen);
    // the core of the application: the finite state machine:
    fsmValve();
    // alarm generation if drought takes too long:
    alarming();
    digitalWrite(VALVE_IO, valveOpen);
    digitalWrite(DRY_ALARM_IO, dryAlarm);
    // wawilib processing of variables:
    WawiSrv.loop();
}
```