Application note: baby oak tree 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 small oak tree based on the moisture level of the soil.

Each year, new oaks grow in my garden as the level of the soil has been raised with ground from a foreign location by the previous owner. The objective of this project is to make sure at least of one of them survives (normally they dry out during summer).

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).

The use of breakpoints in the sketch to debug the application will be documented in this project note.

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.

Fig 1.1. Baby oak tree in garden with moisture sensor.

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 *valveTimeOpenMSec* and *valveTimeHoldMSec*. The both of them contain time values in milliseconds.

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 contains the total 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 *stateString*. The application contains various software timers. WawiLib can be used to read the actual values of these timers using the variables *timerFsm* and *dryAlarmTimer*. 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 1.8.15) 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 fig. 1.2. 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.

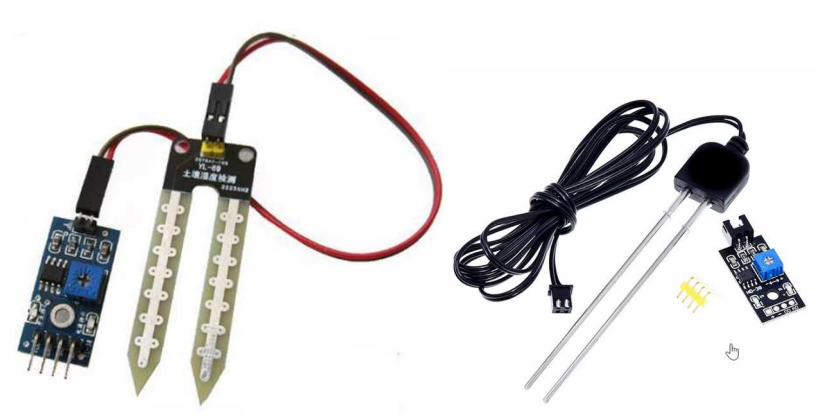


Fig. 1.2. Various types of moisture sensors.

(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.

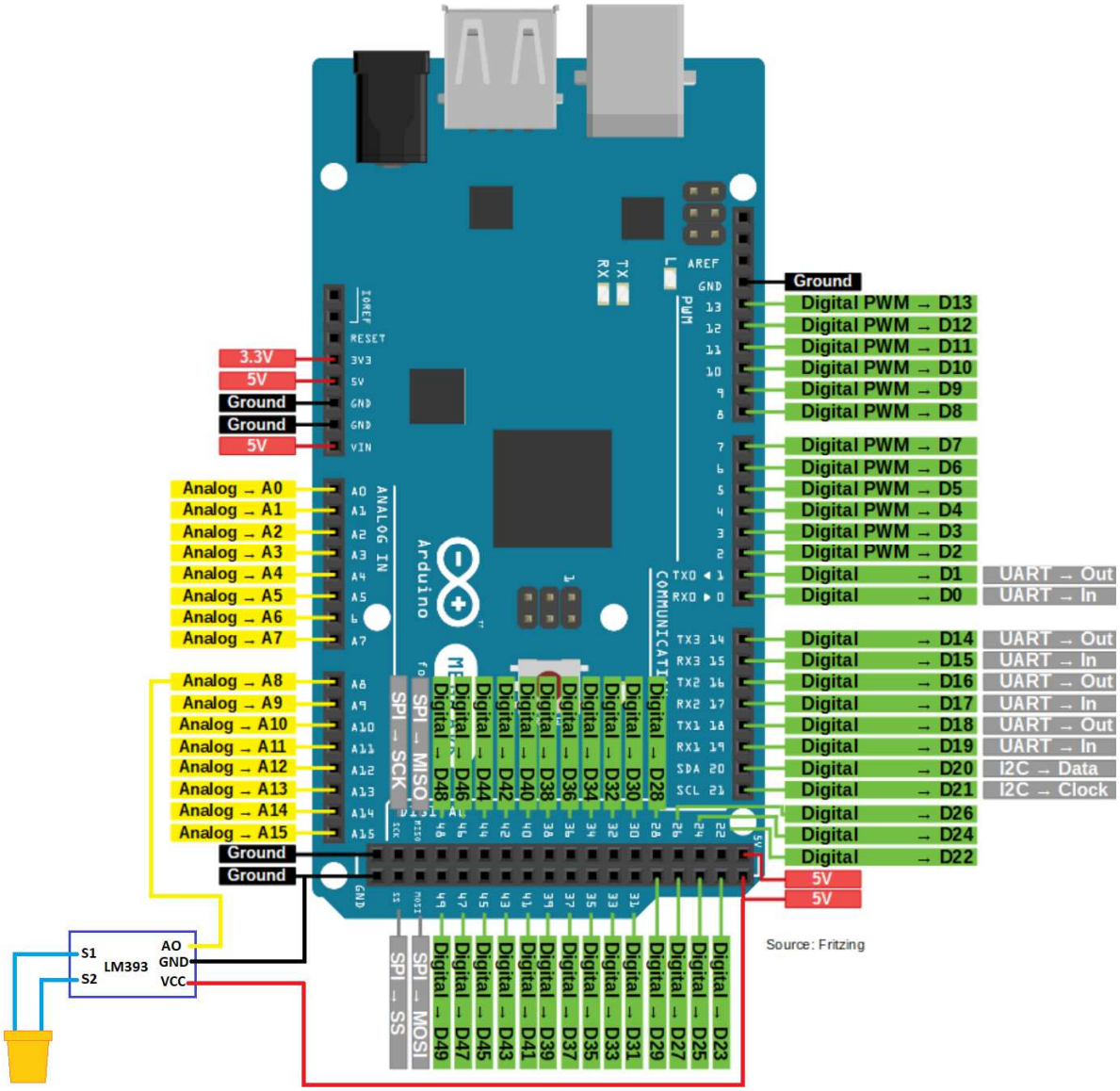


Fig. 2.1. Arduino Mega + sensor connections.

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.

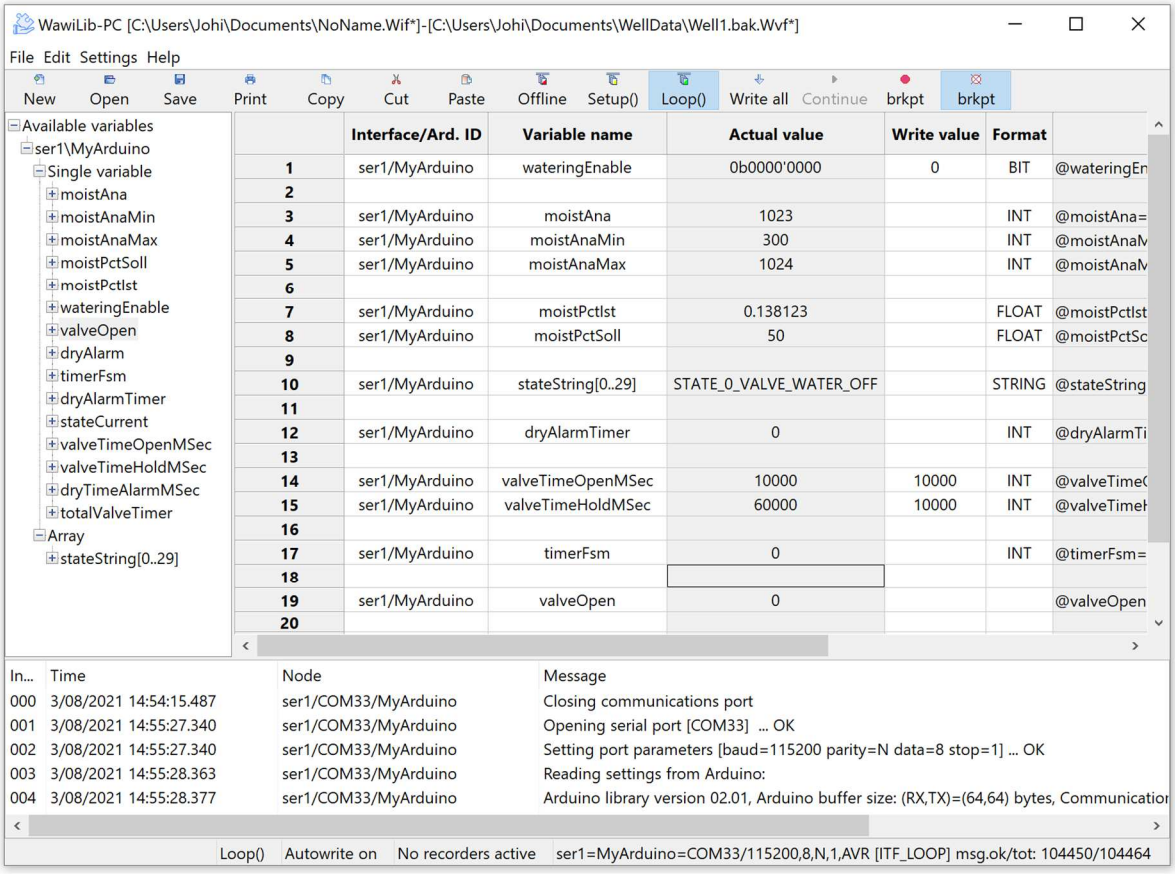


Fig. 2.2. Sketch variabeles of interest.

2.4 Reading and scaling the sensor input

✓ Put the sensor in a glass of water

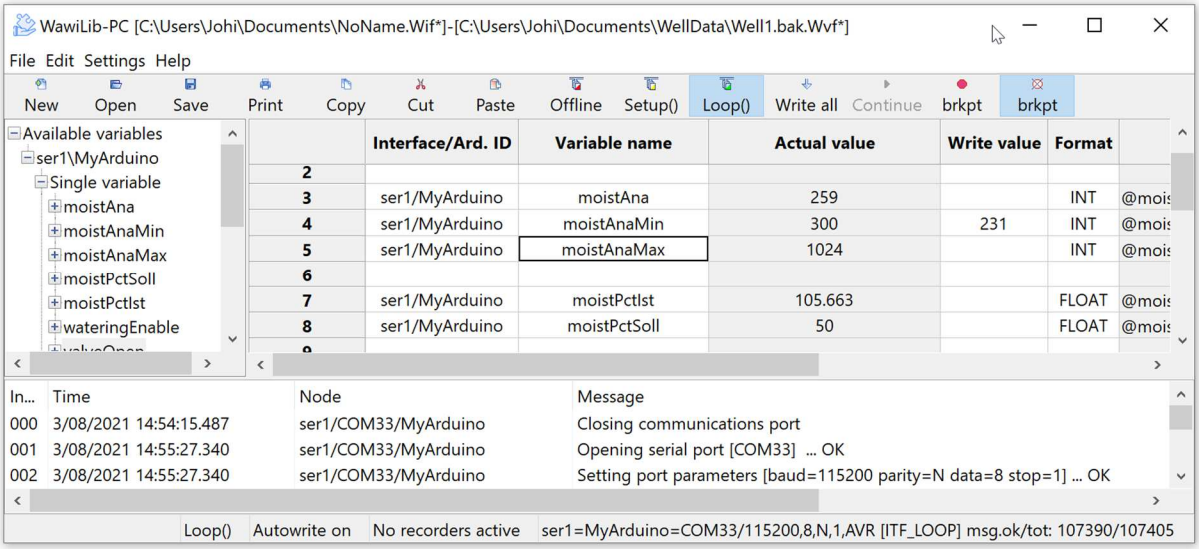


Fig. 2.3. *moistAna* changes when sensor is in glass of water.

⇨ 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 259 in my case (fig 2.3). The output value of the signal converter is about 1.05V (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).

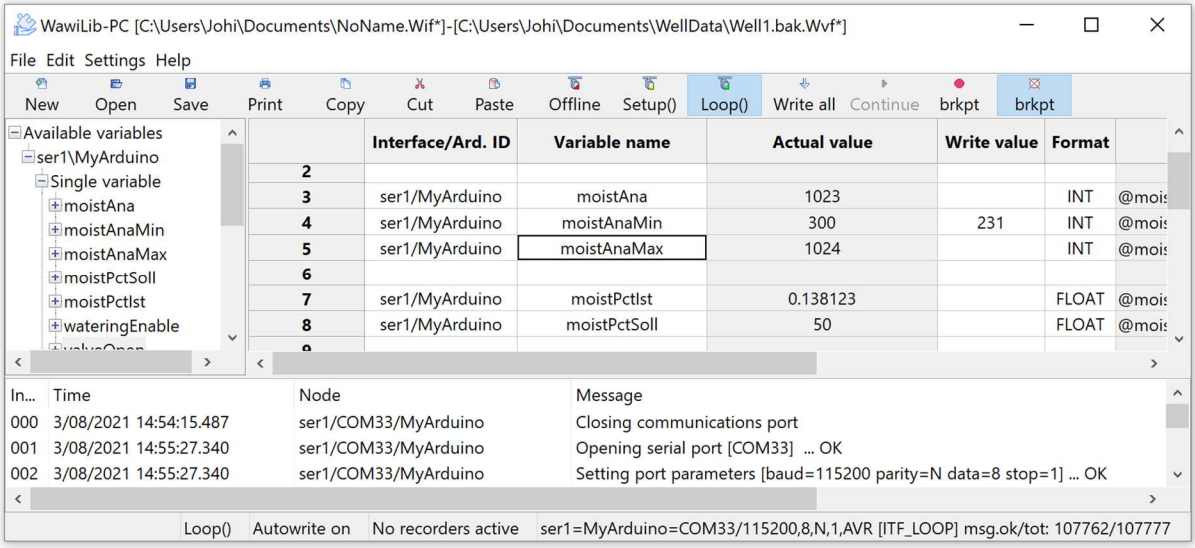


Fig. 2.4. *moistAna* when isolated (not in the water).

⇨ You should see the value of *moistAna* rise (= analog input 0..1023). 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 the *moistAnaMin* and *moistAnaMax* scaling factors. (Use WawiLib to change the value to these parameters.)

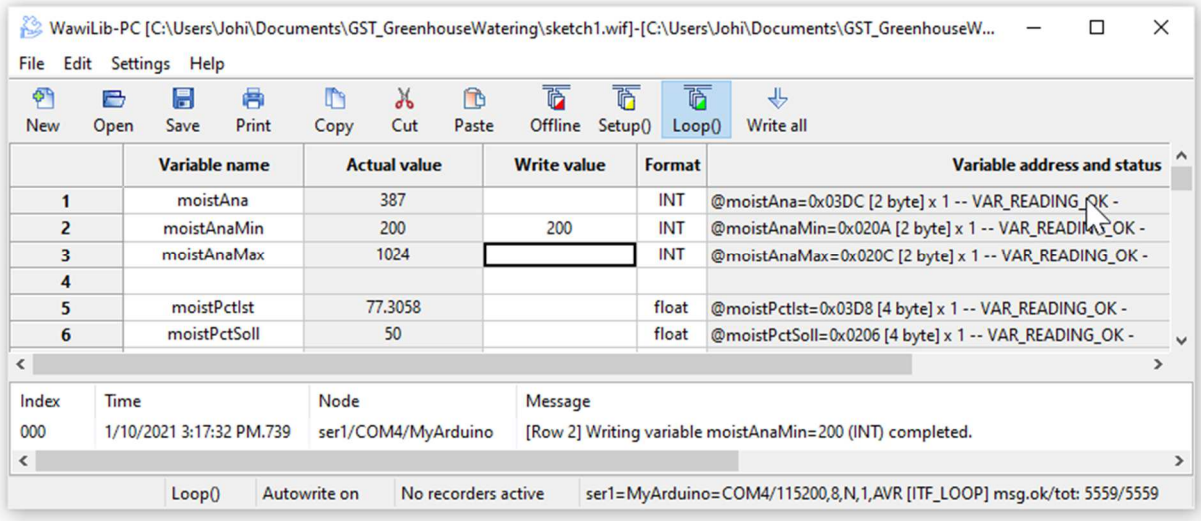


Fig. 2.5. *moistAna* when the sensor is in the water.

⇨ 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 *stateString.*

⇨ Watering is disabled and the FSM (finite state machine) is in the state *"STATE_0_VALVE_WATER_OFF"*.

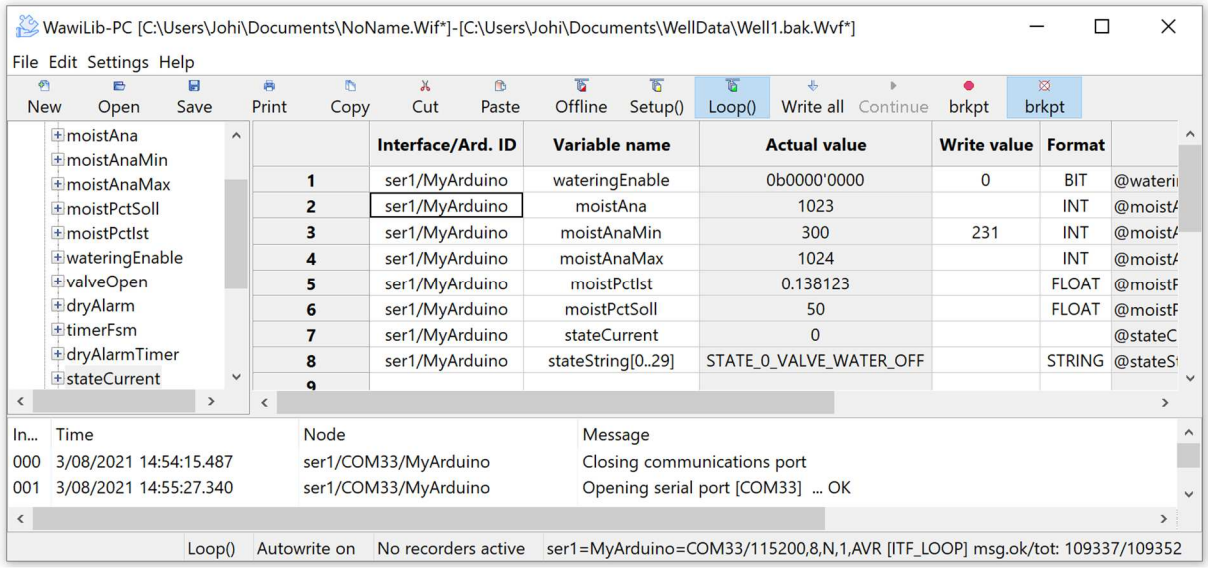


Fig. 2.6. Watering wateringEnable turned off.

✓ Put the sensor in the water

✓ Make sure *moistPctIst* is above *moistPctSoll* (simulate soil wetter than necessary)

✓ Enable "display .print() messages" in the WawiLib output window.

✓ Write the value 1 to the variable *wateringEnable* via WawiLib.

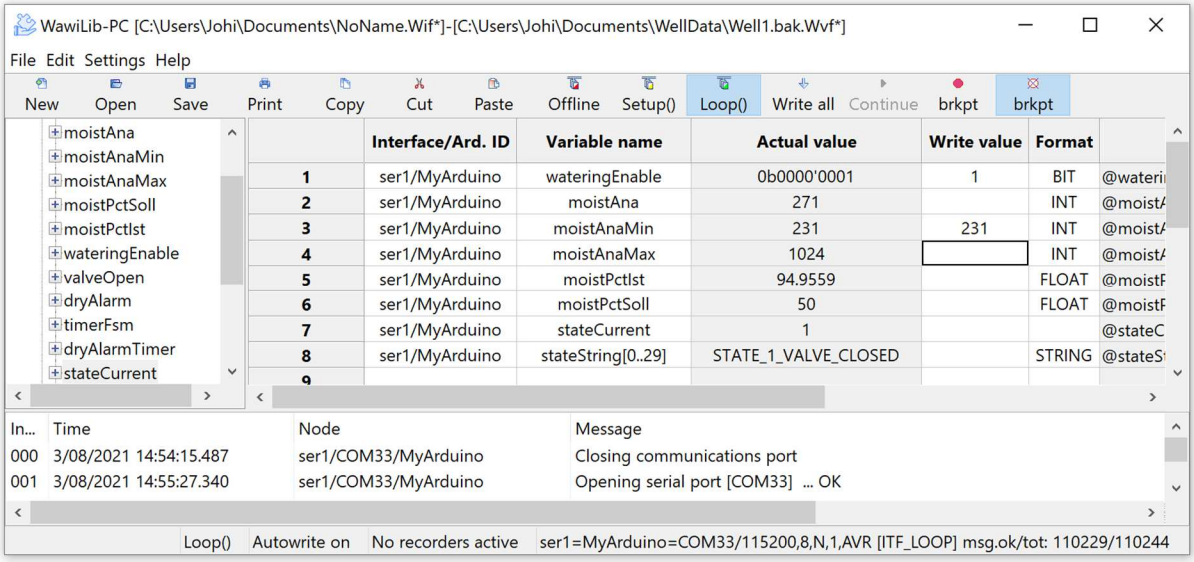


Fig. 2.7. Watering enabled turned on, soil wet enough.

⇨ As watering is enabled now, the FSM changes to the state *"STATE_1_VALVE_CLOSED"*.

✓ Monitor the variables as indicated in the table below.

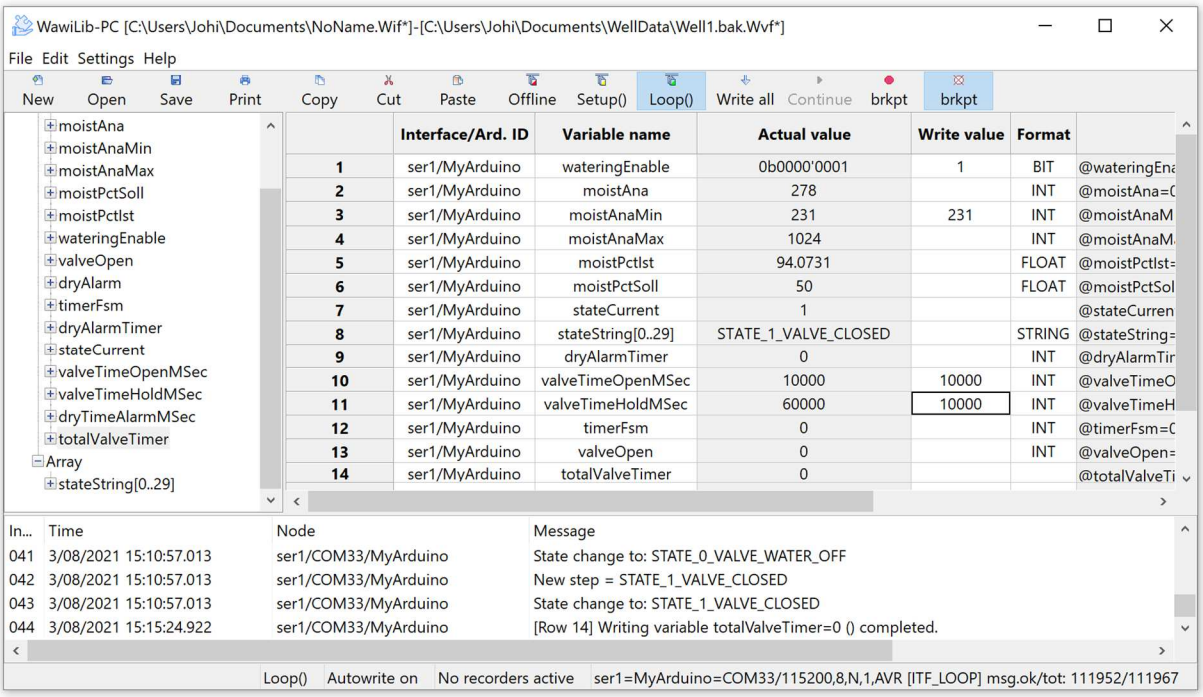
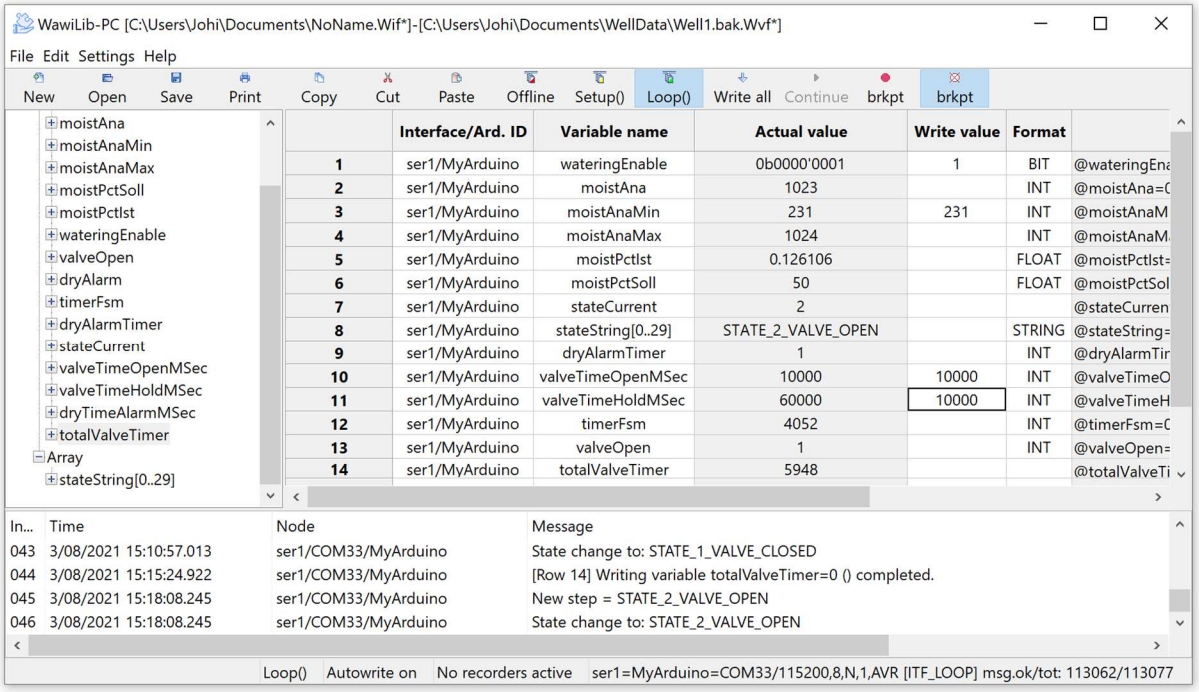


Fig. 2.8. Watering enabled turned on, soil wet enough.

⇨ The state machine is in the state "STATE_1_VALVE_CLOSED" [line 8].
⇨ The valve is closed [line 13].
⇨ totalValveTimerMSec , the total watering time since the boot of the Mega, is 0 [line 12].
⇨ the value of *valveTimeOpenMSec* (the setpoint of the time the valve is open) is 10 seconds [11].
⇨ the value of *valveTimeHoldMSec* (the setpoint of the time the valve is closed to let the water settle) is 60 seconds [11].

2.6 Simulate drought

✓ Remove the sensor from the water.



- ✓ The state machine is in the state “STATE_2_VALVE_OPEN” [line 8].
- ✓ The valve is open [line 13].
- ✓ totalValveTimerMSec , the total watering time since the boot of the Mega, is 5948 msec [line 12].
- ✓ the value of *valveTimeOpenMSec* (the setpoint of the time the valve is open) is 10 seconds [11].
- ✓ the value of *valveTimeHoldMSec* (the setpoint of the time the valve is closed to let the water settle) is 60 seconds [11].
- ✓ Put the sensor back in the water.

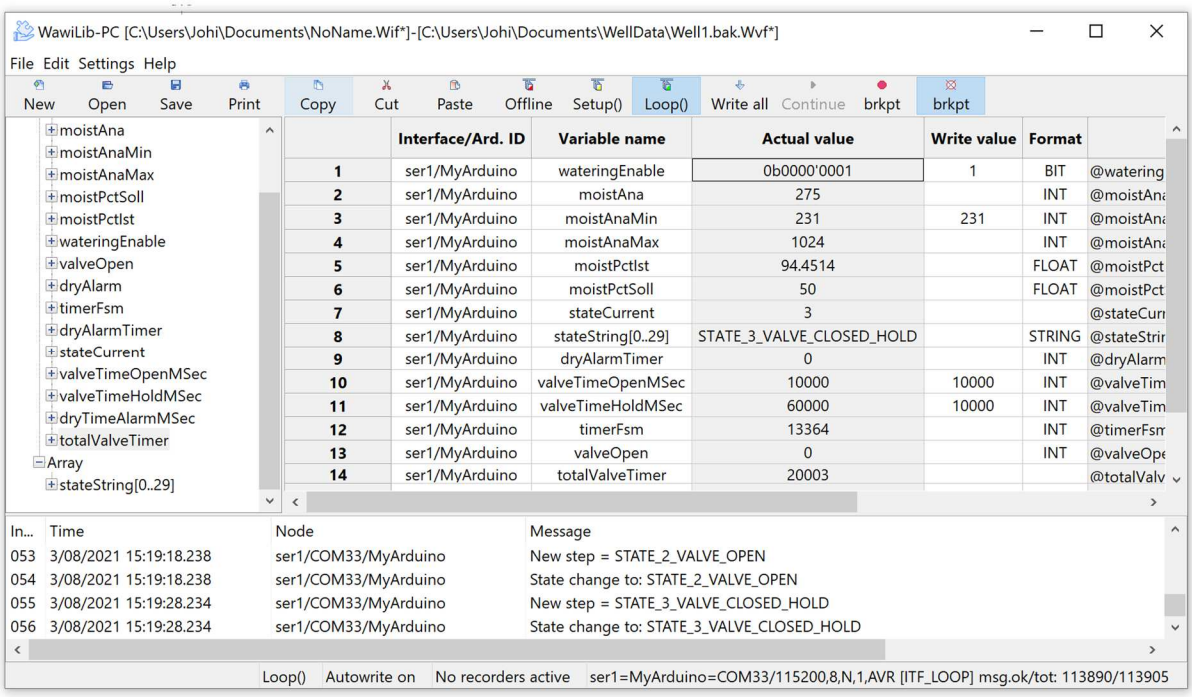


Fig. 2.10. xxx.

⇨ The cycle continues until timerFsm is 0 and then the FSM changes to *“STAE_3_VALVE_CLOSED_HOLD”*

⇨ 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.

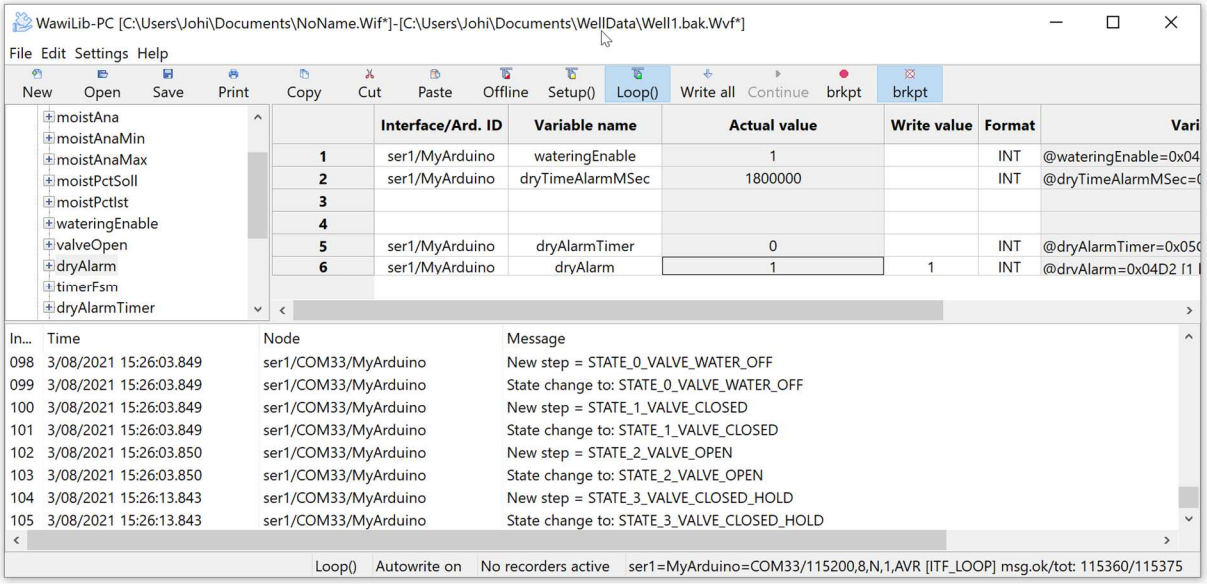


Fig. 2.11. Dry soil alarming.

⇨ The timer *dryAlarmTimer de*creases.

⇨ As it reaches 0, 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. *totalValveTimer* 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):

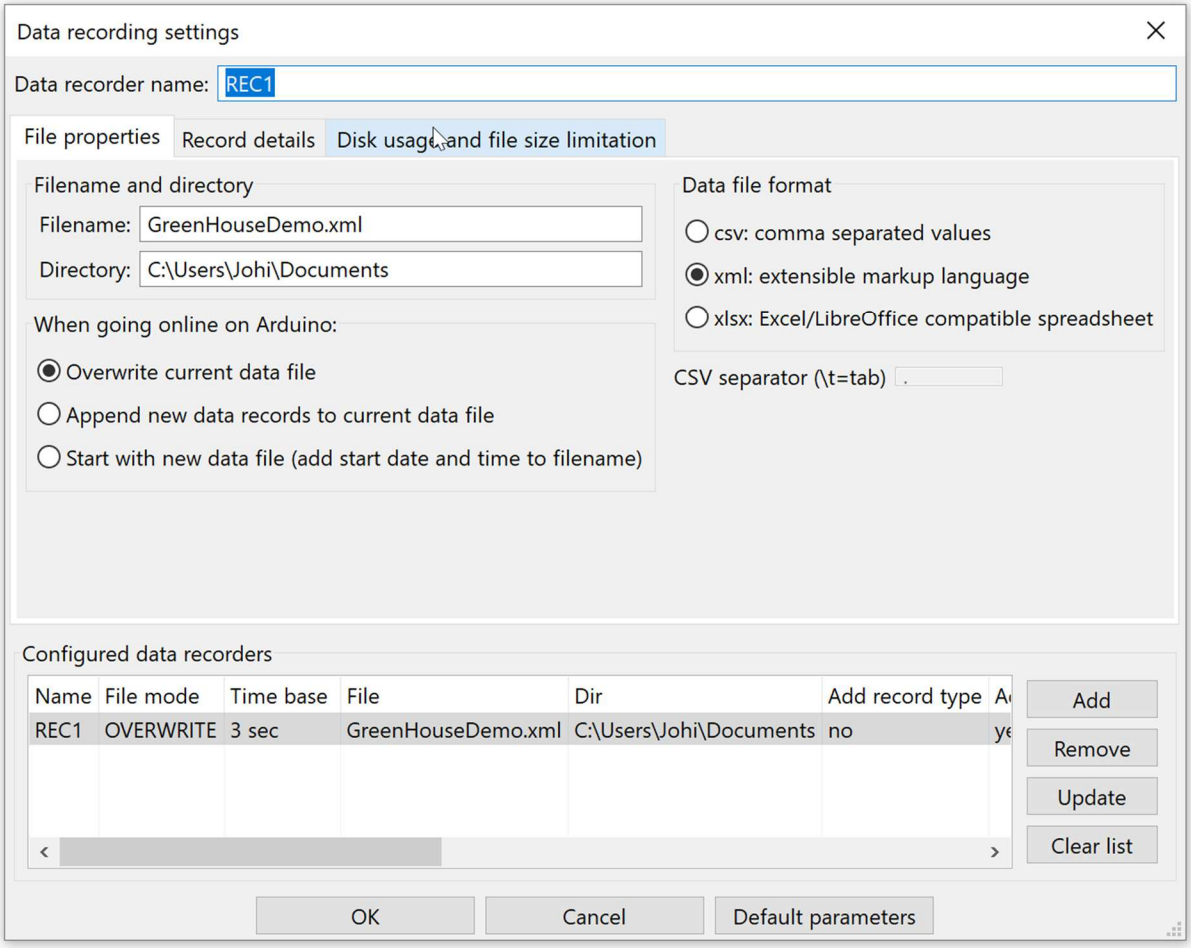


Fig. 3.1. Define a new data recorder.

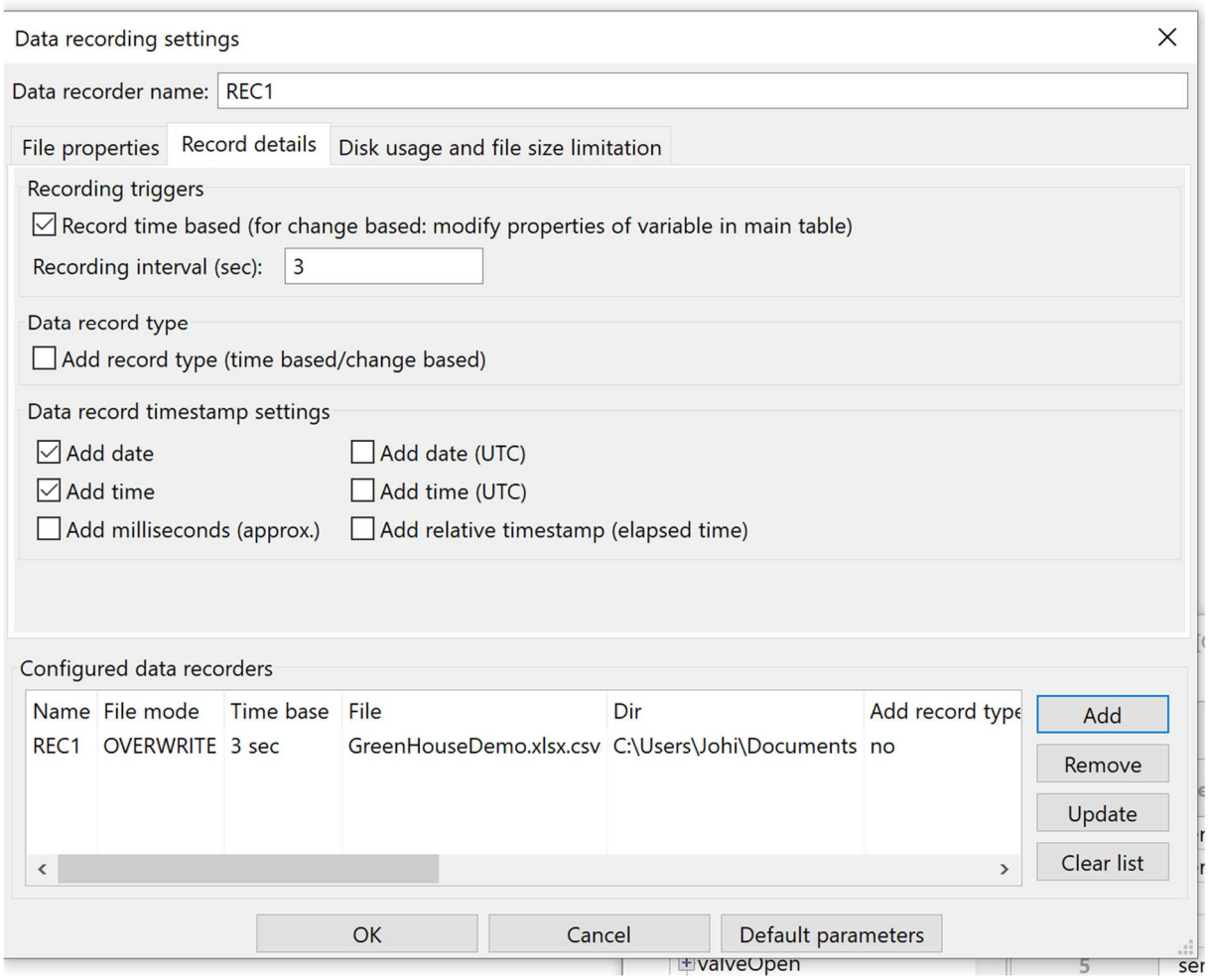


Fig. 3.2. New variable data recorder 3 sec timebase.

- ✓ 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).

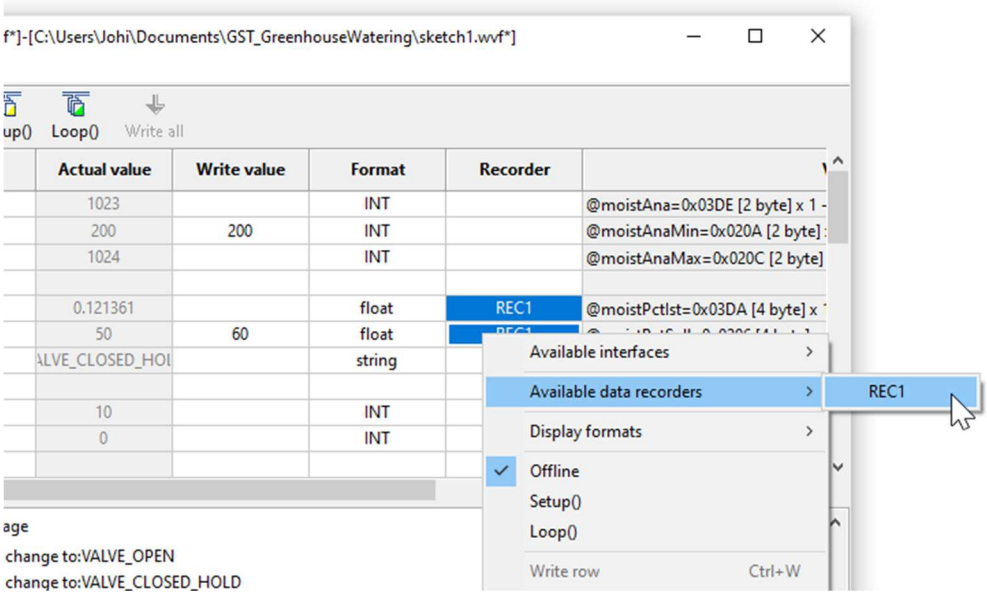


Fig. 3.3. Select data recorder rect 1 for *moistPctIst* and *moistPctSoll*.

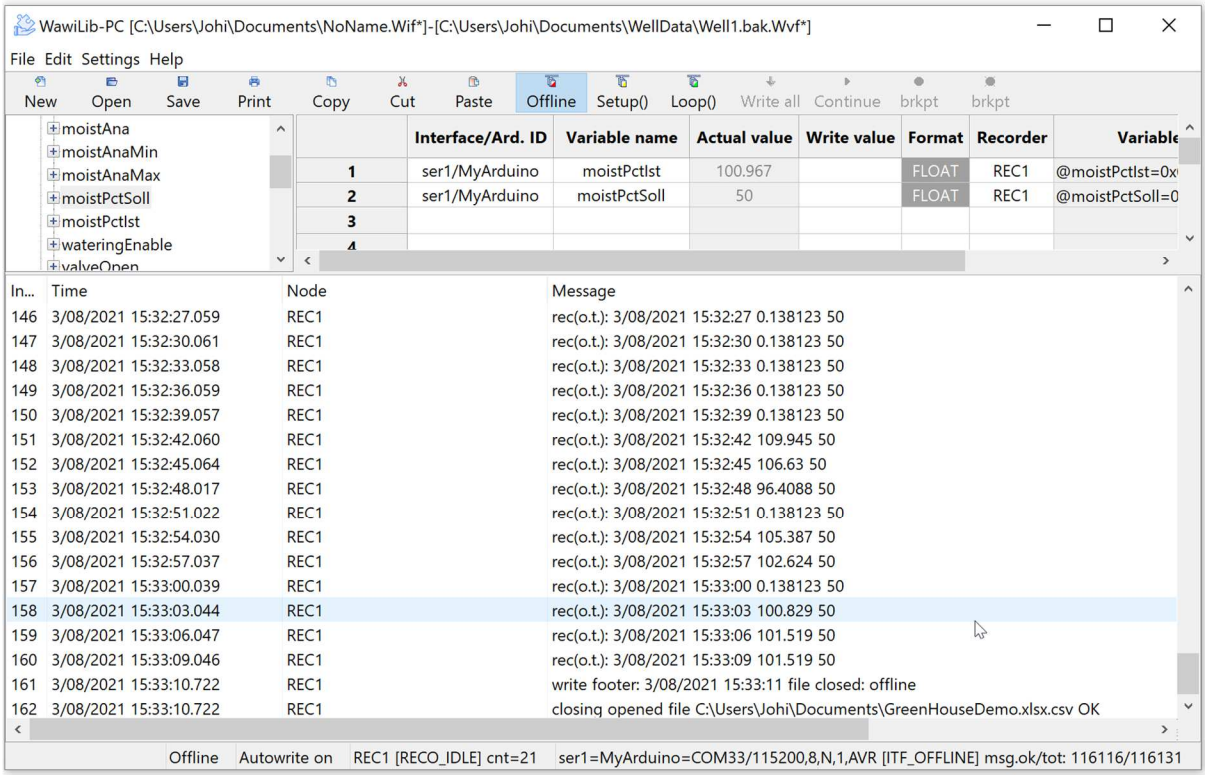
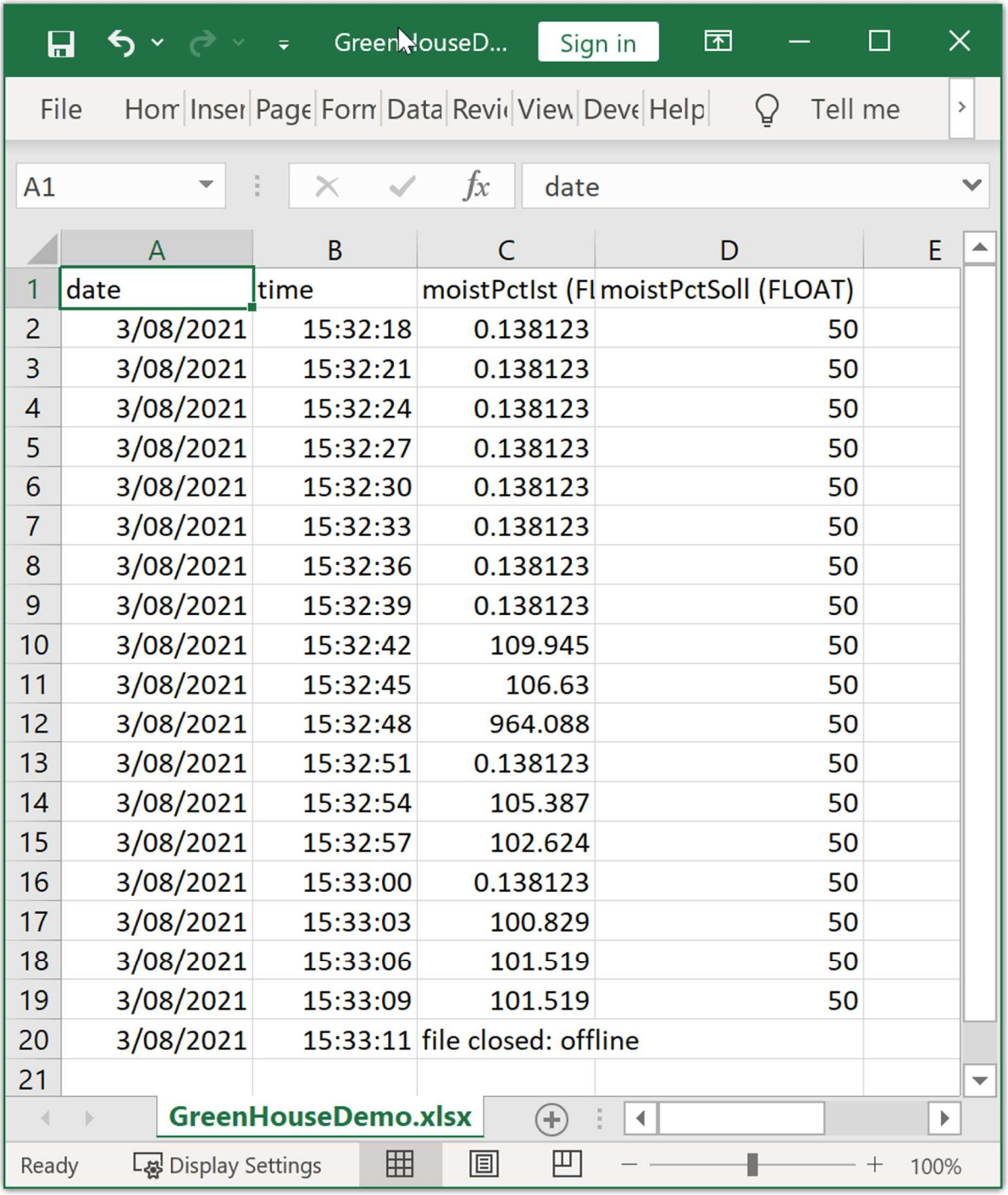


Fig. 3.4. Varying values of *moistPctIst* and static values of *moistPctSoll*.

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



	A	B	C	D	E
1	date	time	moistPctIst (Fl	moistPctSoll (FLOAT)	
2	3/08/2021	15:32:18	0.138123	50	
3	3/08/2021	15:32:21	0.138123	50	
4	3/08/2021	15:32:24	0.138123	50	
5	3/08/2021	15:32:27	0.138123	50	
6	3/08/2021	15:32:30	0.138123	50	
7	3/08/2021	15:32:33	0.138123	50	
8	3/08/2021	15:32:36	0.138123	50	
9	3/08/2021	15:32:39	0.138123	50	
10	3/08/2021	15:32:42	109.945	50	
11	3/08/2021	15:32:45	106.63	50	
12	3/08/2021	15:32:48	964.088	50	
13	3/08/2021	15:32:51	0.138123	50	
14	3/08/2021	15:32:54	105.387	50	
15	3/08/2021	15:32:57	102.624	50	
16	3/08/2021	15:33:00	0.138123	50	
17	3/08/2021	15:33:03	100.829	50	
18	3/08/2021	15:33:06	101.519	50	
19	3/08/2021	15:33:09	101.519	50	
20	3/08/2021	15:33:11	file closed: offline		
21					

Fig. 3.6. Recored data file opened in Microsoft Excel.

⇨ 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 record the values of *stateString* and *valveOpen* (change based):

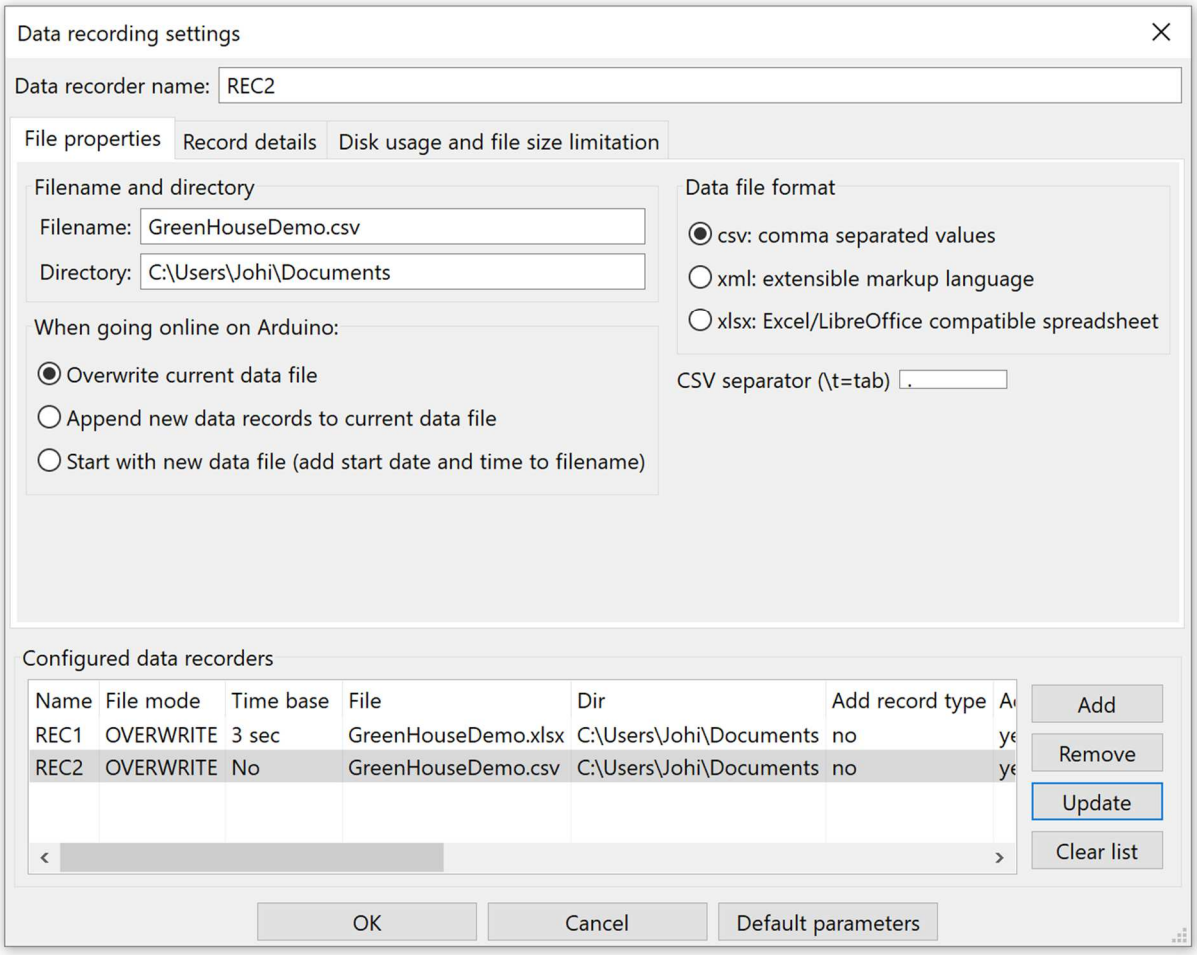


Fig. 3.7. Add a new data recorder REC2, disable time based recording.

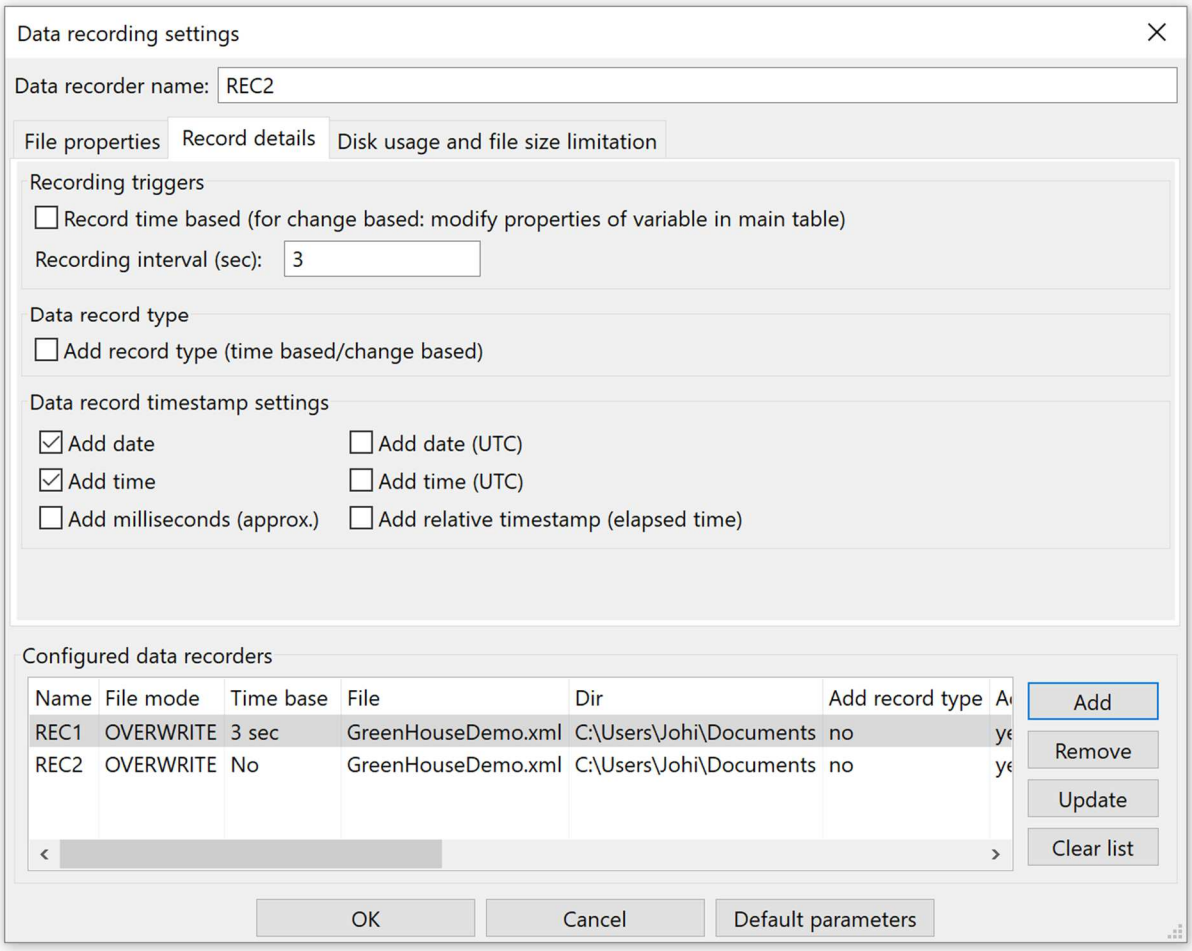


Fig. 3.8. Add a new data recorder REC2, disable time based recording.

✓ Add REC2 as recorder for moistAna, *stateString, and sateCurrent.*
✓ Change the properties of *stateString* as indicated below:

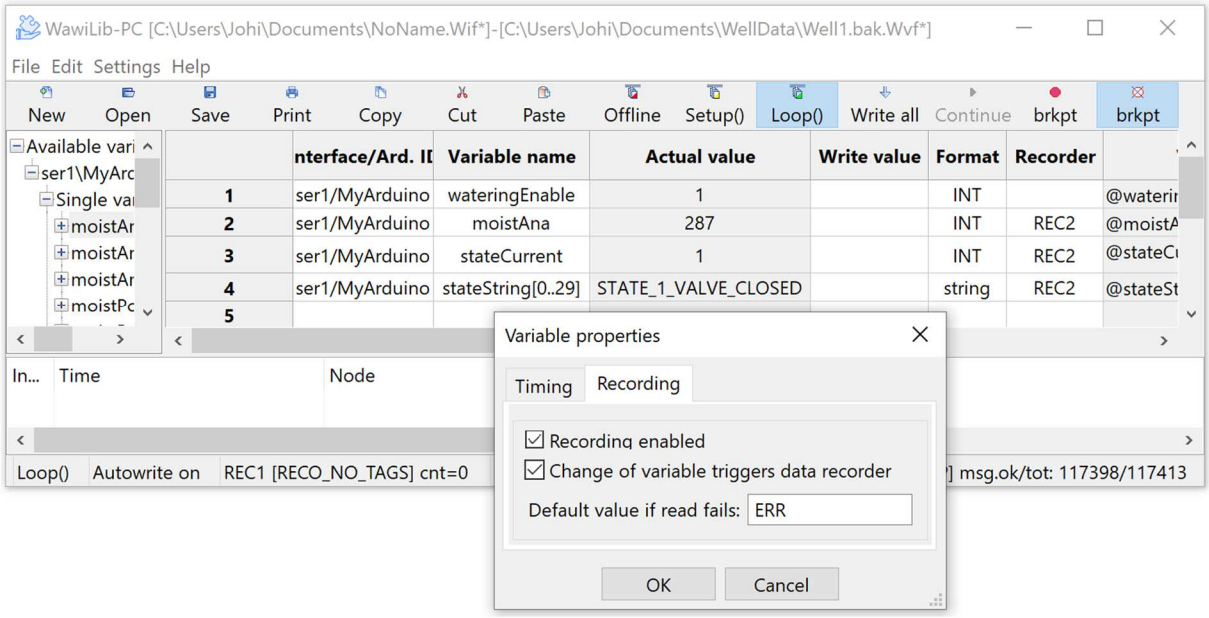
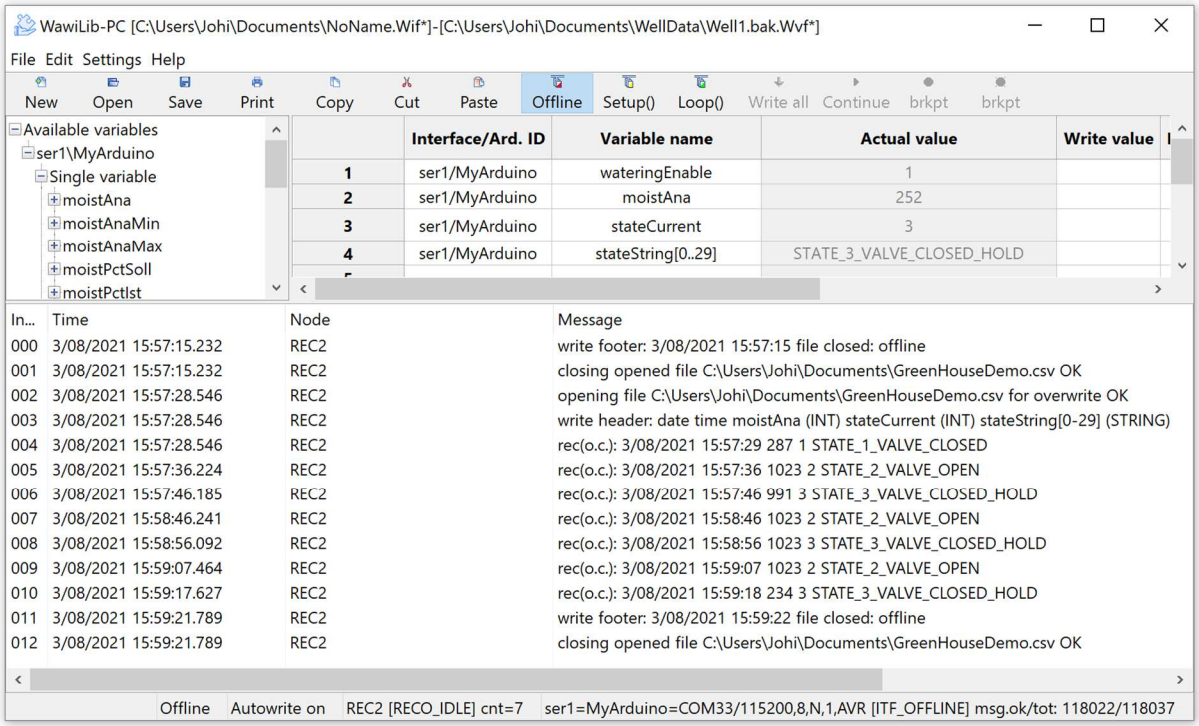


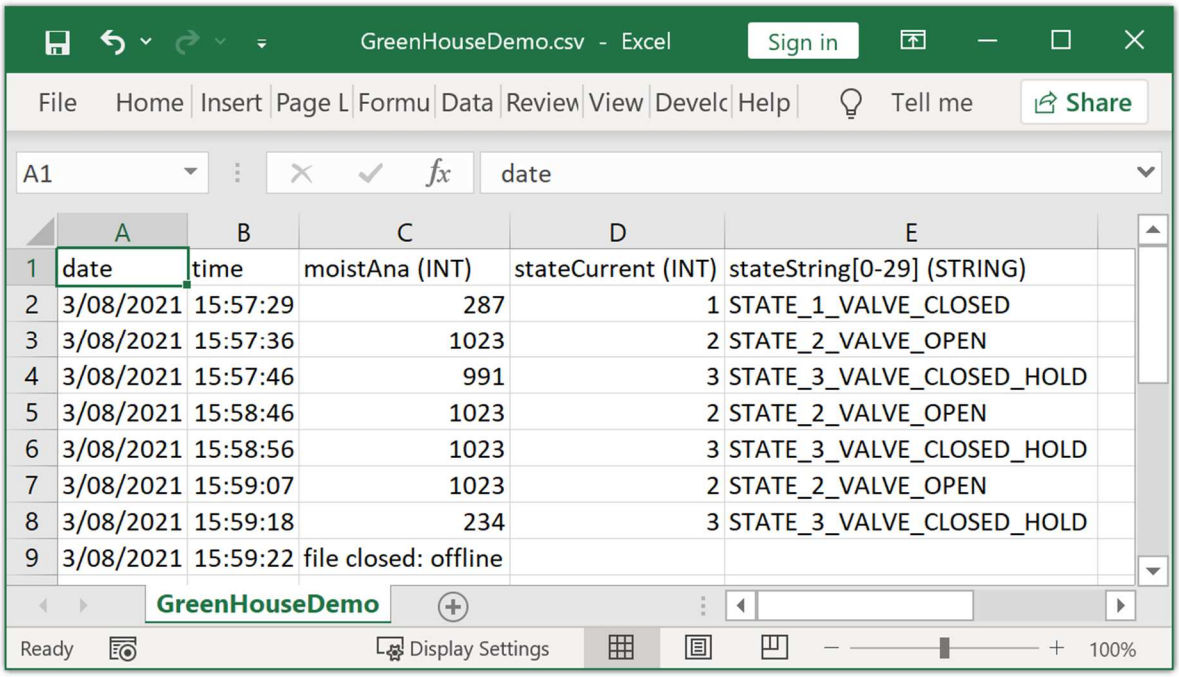
Fig. 3.9. Change the properties of stateString so its changes trigger REC2.

✓ Enable “Display data recording” in the output window.
✓ Delete REC1.
✓ 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.



⇨ Fig. 3.10. A change of the FSM state triggers the write of a new set of data to the disk.

⇨ Once the PC reads a 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.
⇨ You see that other parameters referenced by REC2 are written on disk when the state string changes.
⇨ 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	moistAna (INT)	stateCurrent (INT)	stateString[0-29] (STRING)
2	3/08/2021	15:57:29	287	1	STATE_1_VALVE_CLOSED
3	3/08/2021	15:57:36	1023	2	STATE_2_VALVE_OPEN
4	3/08/2021	15:57:46	991	3	STATE_3_VALVE_CLOSED_HOLD
5	3/08/2021	15:58:46	1023	2	STATE_2_VALVE_OPEN
6	3/08/2021	15:58:56	1023	3	STATE_3_VALVE_CLOSED_HOLD
7	3/08/2021	15:59:07	1023	2	STATE_2_VALVE_OPEN
8	3/08/2021	15:59:18	234	3	STATE_3_VALVE_CLOSED_HOLD
9	3/08/2021	15:59:22	file closed: offline		

Fig. 3.11. recorded data file opened in Microsoft Excel.

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 or in delay

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));
}
```

Fig. 4.1. using delay in a real time program is not a good idea.

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. Therefor 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 WawiTimer object

4.2.1 The concept “millis() without delay in a C++ object”

Arduino’s have a function called *millis().* Millis() returns the number of milliseconds since the Arduino board was booted. If we keep track of the value of millis(), it is possible to create a kind of timer “object” without blocking the main loop.

Suppose, we iterate through the main loop and each time we iterate, we store the value of millis().

By calculating the difference between the current and the previous value of millis(), we know the amount of time that has elapsed since the last pass.

Now, if we keep track of all the time intervals elapsed we know the total time elapsed between an event and the current moment in time. Doing so opens the way for a timer object.

So, first you set the object to a value (10.000 msec), each scan the value of 10 sec is decremented by the time elapsed since the previous scan. We do this until the value of the time counter reaches 0.

Each time we look also at the value of the internal time counter, as long as it is bigger than 0, we do nothing and once is it zero, we start some kind of job or a task. This way we can create a timer functionality without holding the loop. Therefor other jobs that are also managed by the program do not need to be put on hold or frozen.

If you put this functionality in an encapsulated object, you get the object called “WawiTimer”. The object has its own files WawiTimer.cpp and WawiTimer.h.

4.2.2 Short look under the hood.

The first thing you do is

```
WawiTimer timerFsm;
```

Fig. 4.3. WawiTimer class members (WawiTimer.h).

Once you do this a number of member variables are created as declared in the declaration of the C++ object class in the header.

```
class WawiTimer
{
private:
    unsigned long m_tPrev;
    bool m_activePrev;
public:
    WawiTimer(bool incremental = false);
    unsigned long long m_t;
    bool m_incremental;
public:
    bool isZero() const { return (m_t == 0); };
    void setMs(unsigned long long valueMs) { m_t = valueMs; m_tPrev = millis(); };
    void setMsAct(unsigned long valueMs) { m_tPrev = valueMs; };
    unsigned long long getValue() const { return m_t; };
    void loop(bool active = true);
    operator bool() const { return (m_t == 0); };
};
```

Once you call timerFsm.loop() (that is a public member function) the whole principle as declared in the $4.2.1. is executed.

```
void WawiTimer::loop(bool active)
{
    unsigned long m, delta;
    if (!m_activePrev && active)
    {
        m_tPrev = millis();
        m_activePrev = active;
        return;
    }
    // calculate delta time between last loop and this loop;
    if (m_activePrev)
    {
        m = millis();
        delta = m - m_tPrev;
        m_tPrev = m;
    }
    // incrementing  timer counter:
    if (m_incremental && m_activePrev)
    {
        if (m_t <= 0xFFFFFFFFFFFFFFFF - delta)
            m_t = m_t + delta;
        else
            m_t = 0xFFFFFFFFFFFFFFFF;
    }
    // decrementing timer counter:
    if (!m_incremental && m_activePrev)
    {
        if (m_t >= delta)
            m_t = m_t - delta;
        else
            m_t = 0;
    }
    m_activePrev = active;
}
```

Fig. 4.4. WawiTimer internal loop function.

So when the Arduino executes:

timerFsm.isZero(), it executes the code marked in yellow below.

```
class WawiTimer
{
private:
    unsigned long m_tPrev;
    bool m_activePrev;
public:
    WawiTimer(bool incremental = false);
    unsigned long long m_t;
    bool m_incremental;
public:
    bool isZero() const { return (m_t == 0); };
    void setMs(unsigned long long valueMs) { m_t = valueMs; m_tPrev = millis(); };
    void setMsAct(unsigned long valueMs) { m_tPrev = valueMs; };
    unsigned long long getValue() const { return m_t; };
    void loop(bool active = true);
    operator bool() const { return (m_t == 0); };
};
```

Fig. 4.5. WawiTimer class members (WawiTimer.h).

The nice thing about this is that the whole millis() thing is encasulated, (hidden) by the object so it does not trouble the structure the code as many implementations that use millis() directly do.

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, a 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 number of states. It can only change 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.

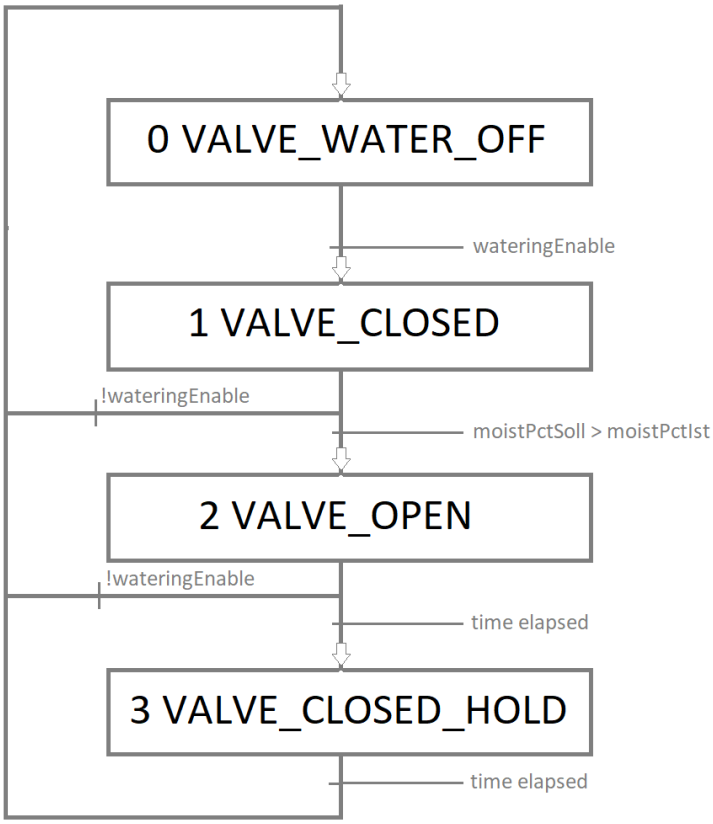


Fig. 4.6. finite state machine of watering application.

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 4.6. you see a visual representation of the finite state machine for the WawiWaterSensorValve program. If the program is not active, the FSM is in the state *STATE_0_VALVE_WATER_OFF*.

In the state *STATE_0_VALVE_WATER_OFF*, the condition *wateringEnable* == true triggers the transition from *STATE_0_VALVE_WATER_OFF* to *STATE_1_VALVE_CLOSED*.

In the state *STATE_1_VALVE_VALVE_CLOSED,* the condition *moistPctSoll > moistPctIst* (actual value of moisture measured is below the setpoint) triggers the transition from *STATE_1_VALVE_WATER_OFF* to *STATE_2_VALVE_OPEN*.

In the state *STATE_2_VALVE_VALVE_OPEN,* the condition timerFsm.isZero() (count down timer has elapsed) triggers the transition to *STATE_3_VALVE_CLOSED_HOLD*.

Other transitions in the diagram work in the same way.

4.3.3 Finite state machine implementation

In C++ programming, an enumeration type (or an enumerator class) is a data type that ideal to represent the states of an FSM. This approach is type safe compared to an approach with an integer number that could obtain an illegal value because of a programming error.

```
// 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 { STATE_0_VALVE_WATER_OFF = 0, STATE_1_VALVE_CLOSED = 1,
STATE_2_VALVE_OPEN = 2, STATE_3_VALVE_CLOSED_HOLD = 3 };
FsmState stateCurrent, statePrev{ FsmState::STATE_0_VALVE_WATER_OFF };
const char* stateStrings[] = { "STATE_0_VALVE_WATER_OFF", "STATE_1_VALVE_CLOSED",
"STATE_2_VALVE_OPEN", "STATE_3_VALVE_CLOSED_HOLD" };
char stateString[30];
bool firstScanNewState;
```

Fig. 4.7. define a enumerator to represent the states of the finite state machine.

If we want to display the actual value in WawiLib, we can use the variable `stateCurrent` and visualize it as an integer number. This is a poor 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 the conversion to a string, I created an array of char* strings *stateStrings[]* and a variable *stateString* that will contain the actual value of the state machine in plain text.

Strcpy copies one of the strings in *stateStrings[]* into *stateString.* The appropriate array index is obtained by using the *stateCurrent* as an integer. (int) is used to cast the Enumto an array index. (fig 4.8.)

```
        // report: translate stateCurrent into a string and put it in stateString:
        strcpy(stateString, stateStrings[(int)stateCurrent]);
```

Fig. 4.8. Convert enum into string containing the actual step.

The code in fig. 4.9. shows the implementation of the FSM itself. The case state using the enum *stateCurrent* makes sure that the code remains transparent. The use of the timer concept explained in the previous chapter is used to implement timer functions.

```
//---------------------------------------------------------------------------
// helper to transfer FSM to other state:
void fsmGoTo(FsmState newState)
{
    stateCurrent = newState;
    WawiSrv.println((String)"New step = " + stateStrings[(int)stateCurrent]);
}
//---------------------------------------------------------------------
// finite state machine:
void loopFsmStateLogic()
{
    do
    {
        // Move from one step to another if conditions are right:
        switch (stateCurrent)
        {
        case FsmState::STATE_0_VALVE_WATER_OFF:
            if (wateringEnable)
                fsmGoTo(FsmState::STATE_1_VALVE_CLOSED);
            break;
        case FsmState::STATE_1_VALVE_CLOSED:
            if (moistPctSoll > moistPctIst)
            {
                WawiSrv.wawiBreak(1, "Break when soil is to dry!");
                fsmGoTo(FsmState::STATE_2_VALVE_OPEN);
            }
            if (!wateringEnable)
                fsmGoTo(FsmState::STATE_0_VALVE_WATER_OFF);
            break;
        case FsmState::STATE_2_VALVE_OPEN:
            if (firstScanNewState)
            {
                WawiSrv.wawiBreak(2, "Break entering step 3.");
                timerFsm.setMs(valveTimeOpenMSec);
            }
            if (timerFsm.isZero())
                fsmGoTo(FsmState::STATE_3_VALVE_CLOSED_HOLD);
            if (!wateringEnable)
                fsmGoTo(FsmState::STATE_0_VALVE_WATER_OFF);
            break;
        case FsmState::STATE_3_VALVE_CLOSED_HOLD:
            if (firstScanNewState)
            {
                WawiSrv.wawiBreak(3, "Break entering step 4.");
                timerFsm.setMs(valveTimeHoldMSec);
            }
            if (timerFsm.isZero())
                fsmGoTo(FsmState::STATE_0_VALVE_WATER_OFF);
            break;
        }
        firstScanNewState = (stateCurrent != statePrev);
        // report: translate stateCurrent into a string and put it in stateString:
        strcpy(stateString, stateStrings[(int)stateCurrent]);
        // display state change of the FSM in the output window:
        if (firstScanNewState)
        {
            statePrev = stateCurrent;
            WawiSrv.print("State change to: ");
```

```
            WawiSrv.println(stateString);
        }
    } while (firstScanNewState);
}
```

Fig. 4.9. the actual finite state machine implementation using a switch and case statements.

Each time the FSM goes from one state to another, the variable *firstScanNewState* is set. Upon entry of the Case section related to the new state, firstScanNewState is used to initialize timerFsm with the target value.

Then, the program keeps circulating through the case statement related to the current step loop after loop. This until (for example) *timerFsm.isZero()* returns true.

If a transfer condition becomes true, the FSM changes state to a new active state using the function *fsmGoTo()*.

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.

```
//------------------------------------------------------------------
// finite state machine:
void loopFsmStateToOutputs()
{
    // Move from one step to another if conditions are right:
    switch (stateCurrent)
    {
    case FsmState::STATE_0_VALVE_WATER_OFF:
    case FsmState::STATE_1_VALVE_CLOSED:
    case FsmState::STATE_3_VALVE_CLOSED_HOLD:
        valveOpen = LOW;
        break;
    case FsmState::STATE_2_VALVE_OPEN:
        valveOpen = HIGH;
        break;
    }
}
```

Fig. 4.10. FSM deterimnes the status of the IO's.

In the same way as for the transitions, you can use a switch() statement to control the states of the FMS output. Putting the control of the outputs in a separate function keeps the code clean and easy to manage.

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
* Updated August 2021
* More info: www.sylvestersolutions.com
* Technical support: support@sylvestersolutions.com
* Additional info: info@sylvestersolutions.com
*/

#include <WawiSerialUsb.h>
#include "WawiTimer.h"

// comment next line out (use //) if you do not have IO present and want to use
WawiLib to simulate IO
#define ENABLE_IO 1

#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 %

//-----------------------------------------------------------------------
// wateringEnable 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;

WawiTimer timerFsm;
WawiTimer dryAlarmTimer();
WawiTimer totalValveTimer(/*incrementing */ true);

//-----------------------------------------------------------------------
// alarm delay time:
unsigned long long dryTimeAlarmMSec = 1800000;

//-----------------------------------------------------------------------
// 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:
long long valveTimeOpenMSec = 10000;
long long valveTimeHoldMSec = 60000;

//-----------------------------------------------------------------------
// 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 { STATE_0_VALVE_WATER_OFF = 0, STATE_1_VALVE_CLOSED = 1,
STATE_2_VALVE_OPEN = 2, STATE_3_VALVE_CLOSED_HOLD = 3 };
FsmState stateCurrent, statePrev{ FsmState::STATE_0_VALVE_WATER_OFF };
const char* stateStrings[] = { "STATE_0_VALVE_WATER_OFF", "STATE_1_VALVE_CLOSED",
"STATE_2_VALVE_OPEN", "STATE_3_VALVE_CLOSED_HOLD" };
char stateString[30];
bool firstScanNewState;

//-----------------------------------------------------------------
// 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(dryTimeAlarmMSec);
    WawiSrv.wawiVar(valveTimeOpenMSec);
    WawiSrv.wawiVar(valveTimeHoldMSec);

    WawiSrv.wawiVar(valveOpen);
    WawiSrv.wawiVar(dryAlarm);

    WawiSrv.wawiVar(stateCurrent);

    WawiSrv.wawiVarTimer(timerFsm);
    WawiSrv.wawiVarTimer(dryAlarmTimer);
    WawiSrv.wawiVarTimer(totalValveTimer);

    WawiSrv.wawiVarArray(stateString);
}

//-----------------------------------------------------------------------------
------
void setupIO()
{
    pinMode(VALVE_IO, OUTPUT);
    pinMode(DRY_ALARM_IO, OUTPUT);
}
//-----------------------------------------------------------------------------
------
void setupWawiLib()
{
    WawiSrv.begin(wawiVarDef, Serial, "MyArduino");
    WawiSrv.wawiBreakDisable();
}
//-----------------------------------------------------------------------------
------
void setup()
{
    Serial.begin(115200);
    setupWawiLib();
    setupIO();
}
```

```
//============================================================================================
======
// 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);
}

//--------------------------------------------------------------------
void loopReadInputs()
{
#ifdef ENABLE_IO
    // read and process moisture level:
    moistAna = limit(analogRead(8), 0, 1024);
    moistPctIst = scale(moistAna, moistAnaMin, moistAnaMax, 100.0, 0.0);
#endif
}

//--------------------------------------------------------------------------------------------
------
// helper to transfer FSM to other state:
void fsmGoTo(FsmState newState)
{
    stateCurrent = newState;
    WawiSrv.println((String)"New step = " + stateStrings[(int)stateCurrent]);
}

//--------------------------------------------------------------------
// finite state machine:
void loopFsmStateLogic()
{
    do
    {
        // Move from one step to another if conditions are right:
        switch (stateCurrent)
        {
        case FsmState::STATE_0_VALVE_WATER_OFF:
            if (wateringEnable)
                fsmGoTo(FsmState::STATE_1_VALVE_CLOSED);
            break;
        case FsmState::STATE_1_VALVE_CLOSED:
            if (moistPctSoll > moistPctIst)
            {
                WawiSrv.wawiBreak(1, "Break when soil is to dry!");
                fsmGoTo(FsmState::STATE_2_VALVE_OPEN);
            }
            if (!wateringEnable)
                fsmGoTo(FsmState::STATE_0_VALVE_WATER_OFF);
            break;
        case FsmState::STATE_2_VALVE_OPEN:
            if (firstScanNewState)
            {
                WawiSrv.wawiBreak(2, "Break entering step 3.");
                timerFsm.setMs(valveTimeOpenMSec);
            }
```

```
            if (timerFsm.isZero())
                fsmGoTo(FsmState::STATE_3_VALVE_CLOSED_HOLD);
            if (!wateringEnable)
                fsmGoTo(FsmState::STATE_0_VALVE_WATER_OFF);
            break;
        case FsmState::STATE_3_VALVE_CLOSED_HOLD:
            if (firstScanNewState)
            {
                WawiSrv.wawiBreak(3, "Break entering step 4.");
                timerFsm.setMs(valveTimeHoldMSec);
            }
            if (timerFsm.isZero())
                fsmGoTo(FsmState::STATE_0_VALVE_WATER_OFF);
            break;
        }
        firstScanNewState = (stateCurrent != statePrev);
        // report: translate stateCurrent into a string and put it in stateString:
        strcpy(stateString, stateStrings[(int)stateCurrent]);
        // display state change of the FSM in the output window:
        if (firstScanNewState)
        {
            statePrev = stateCurrent;
            WawiSrv.print("State change to: ");
            WawiSrv.println(stateString);
        }
    } while (firstScanNewState);
}

//---------------------------------------------------------------------
// Make alarm if dry is too long high:
void loopAlarming()
{
    // if watering is wateringEnabled, and moisture is too low a timer is started
    bool dry = moistPctIst < moistPctSoll;
    if (wateringEnable && !dry)
        dryAlarmTimer.setMs(dryAlarmTimer);

    // alarm after moistureTimeAlarmSec seconds of drought
    if (dryAlarmTimer.isZero())
        dryAlarm = HIGH;
    else
        dryAlarm = LOW;
}

//---------------------------------------------------------------------
// finite state machine:
void loopFsmStateToOutputs()
{
    // Move from one step to another if conditions are right:
    switch (stateCurrent)
    {
    case FsmState::STATE_0_VALVE_WATER_OFF:
    case FsmState::STATE_1_VALVE_CLOSED:
    case FsmState::STATE_3_VALVE_CLOSED_HOLD:
        valveOpen = LOW;
        break;
    case FsmState::STATE_2_VALVE_OPEN:
        valveOpen = HIGH;
        break;
    }
}
```

```
//---------------------------------------------------------------------
void loopWriteOutputs()
{
    digitalWrite(VALVE_IO, valveOpen);
    digitalWrite(DRY_ALARM_IO, dryAlarm);
}

void loop()
{
    loopReadInputs();
    loopFsmStateLogic();
    loopAlarming();
    loopFsmStateToOutputs();
    timerFsm.loop();
    totalValveTimer.loop(valveOpen);
    dryAlarmTimer.loop();
    WawiSrv.loop();
}
```

Fig 6.1. WawiWaterSensorValve.ino

```
#include <Arduino.h>

#define wawiVarTimer(X)  wawiCompare(#X,(unsigned int) &X.m_t,sizeof(X.m_t),1);

class WawiTimer
{
private:
    unsigned long m_tPrev;
    bool m_activePrev;
public:
    WawiTimer(bool incremental = false);
    unsigned long long m_t;
    bool m_incremental;
public:
    bool isZero() const { return (m_t == 0); };
    void setMs(unsigned long long valueMs) { m_t = valueMs; m_tPrev = millis(); };
    void setMsAct(unsigned long valueMs) { m_tPrev = valueMs; };
    unsigned long long getValue() const { return m_t; };
    void loop(bool active = true);
    operator bool() const { return (m_t == 0); };
};
```

Fig 6.2. WawiTimer.h

```
#include "WawiTimer.h"
#include "Arduino.h"

WawiTimer::WawiTimer(bool incremental)
{
    m_incremental = incremental;
    m_t = 0;
}

void WawiTimer::loop(bool active)
{
    unsigned long m, delta;
    if (!m_activePrev && active)
    {
        m_tPrev = millis();
        m_activePrev = active;
        return;
    }
    // calculate delta time between last loop and this loop;
```

```
    if (m_activePrev)
    {
        m = millis();
        delta = m - m_tPrev;
        m_tPrev = m;
    }
    // incrementing  timer counter:
    if (m_incremental && m_activePrev)
    {
        if (m_t <= 0xFFFFFFFFFFFFFFFF - delta)
            m_t = m_t + delta;
        else
            m_t = 0xFFFFFFFFFFFFFFFF;
    }
    // decrementing timer counter:
    if (!m_incremental && m_activePrev)
    {
        if (m_t >= delta)
            m_t = m_t - delta;
        else
            m_t = 0;
    }
    m_activePrev = active;
}
```

Fig 6.3. WawiTimer.cpp