Eksperymentalny, pojemnościowy, mówiący czujnik wilgotności gleby

Pomiar wilgotności gleby wydaje się zadaniem trywialnym – wystarczy zmierzyć jej rezystancję. Niestety, w ten sposób zmierzymy tylko jej przewodność, zależną też od zawartości soli mineralnych (nawozów). Ponadto prąd przepływający przez elektrody powoduje ich korozję. Pomiar prądem przemiennym oraz wykonanie sond z materiałów odpornych na korozję nie likwidują problemu. Warto wiedzieć, że można oszacować wilgotność przez pomiar pojemności, gdzie gleba jest dielektrykiem między okładkami kondensatora.

Opisywane urządzenie służy do wykrywania zmiany wilgotności gleby. Zaprezentowane w artykule rozwiązanie służy do sygnalizacji suchej gleby w doniczce. Gdy wilgotność spadnie poniżej ustalonego poziomu, na wyjściu mikrokontrolera będzie odtwarzany krótki plik dźwiękowy. Urządzenie, a raczej metoda pomiarowa, może też służyć do pomiaru poziomu płynu, a to dzięki liniowym zmianom pojemności sondy w zależności od poziomu płynu. Pojemnościowy pomiar pozwala użyć izolowanych sond, dzięki czemu nie korodują. Elektrody mogą być zamontowane na zewnątrz pojemnika z cieczą. W najprostszym rozwiązaniu mogą to być dwa paski folii aluminiowej naklejonej na zewnętrznej stronie naczynia.

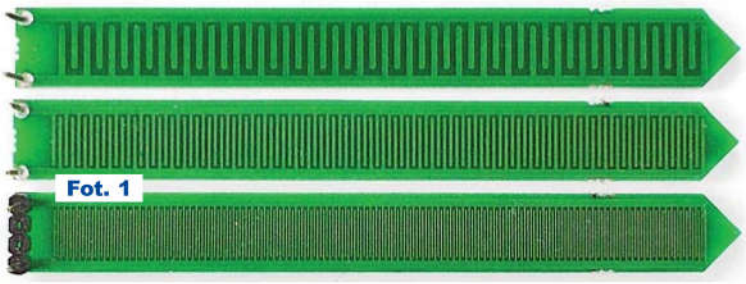
Opis układu

Mimo zbudowania użytecznego urządzenia, artykuł ma charakter edukacyjny. Zanim powstała prototypowa konstrukcja, została zbadana zmiana pojemności gleby względem wilgotności. Do prób powstały trzy sondy o różnej liczbie zębów „grzebienia" wykonanego ze ścieżek na PCB, jak pokazuje **fotografia 1**. Zmierzona została pojemność każdej z sond dla trzech przypadków. Wyniki zestawiono w **tabeli 1**.

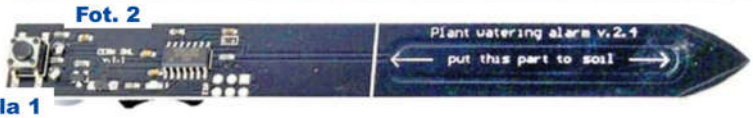
Wyniki nie są jednoznaczne. Wątpliwości budzą wyniki dotyczące sondy numer 2, dlatego testy przeprowadziłem trzykrotnie. Wyniki zawsze były podobne. Uznałem, że jest jakiś problem z sondą o średniej gęstości i do dalszych testów wybrałem sondę o małej pojemności. Być może jeszcze lepsze wyniki udałoby się uzyskać, stosując sondę w postaci dwóch równoległych ścieżek, ale nie zamawiałem kolejnych PCB z sondami o różnych konstrukcjach. Jeśli Czytelnicy wykażą zainteresowanie, przeprowadzę kolejne testy.

Gdy już znany był zakres zmian pojemności, należało zastanowić się, jaką metodę pomiarową wybrać. Do testów został zakupiony gotowy czujnik (**fotografia 2**), który okazał się w praktyce nieużyteczny, ponieważ cały czas sygnalizował suchą glebę – być może w Chinach gleba ma inny skład chemiczny ;) W internecie znalazłem schemat czujnika wilgotności, przedstawiony na **rysunku 1**. Na schemacie widać obwód generujący sygnał prostokątny, który przez rezystor ładuje i rozładowuje badaną pojemność. Obwód złożony z diody, kondensatora i rezystora tworzy prostownik z filtrem. Miałem wątpliwości co do skuteczności tej metody w przypadku małych pojemności (spadek napięcia na diodzie), dlatego zastosowałem nieco inne rozwiązanie. Pomiar polega na pomiarze czasu ładowania pojemności sondy przez rezystor 1 MΩ. Pierwotnie do tego celu chciałem użyć mikrokontrolera AVR ATtiny13. Okazało się jednak, że tańszy i dużo lepszy jest ATtiny85, którego jedną z zalet jest interfejs debuggera. Przez myśl przeszedł mi pomysł, że w 8kB FLASH można zmieścić sekundę nagrania jakości „telefonicznej" 8-bit 4kHz, dzięki czemu czujnik mógłby przemówić.

W telefonii powszechnie używane jest kodowanie G711u/aLaw, w którym próbki 12-bit zbierane z częstotliwością 8kHz (pasmo 4kHz) są kompresowane do 8-bit, po czym u odbiorcy dekompresowane z 8 na 12-bit. Pozwala to uzyskać większy odstęp sygnału od szumu podobnie jak kompresja dynamiki w radiu FM (preemfaza i deemfaza) czy systemach redukcji szumu Dolby stosowanego przy zapisie dźwięku na taśmie magnetycznej.



Fot. 1



Fot. 2

Tabela 1

Typ sondy	Pojemność własna sondy	Gleba sucha	Gleba mokra	Różnica mokra – sucha	Stosunek zmian (mokra-sucha)/ pojemność własna
rzadka	50pF	204pF Δ+154	330pF Δ+280	126pF	252%
średnia	70pF	175pF Δ+105	385pF Δ+315	210pF	300%
gęsta	120pF	288pF Δ+108	507pF Δ+387	279pF	232%

Sekunda to niewiele a i program musi zająć część pamięci FLASH, szacunkowo ok. 1kB. Zacząłem szukać tanich mikrokontrolerów ARM i znalazłem **STM32G030,** który można kupić w obudowie 8-pin i **jest nieznacznie tańszy od Tiny85!** Zasoby, w porównaniu do Tiny85, są naprawdę imponujące. Z najważniejszych należy wymienić dużą pamięć FLASH (32kB) i 8kB RAM (tyle, co FLASH w Tiby85!). Dokładniejsze zestawienie znajduje się w **tabeli 2**.

Po krótkiej reklamie STM32 opiszę zasadę działania programu, którego kody źródłowe są dostępne w materiałach dodatkowych. Fazę pomiaru rozpoczyna rozładowanie pojemności sondy:

```
//----- Przygotowanie pomiaru - Rozładowanie pojemności -----//
 HAL_GPIO_WritePin( GPIOA, GPIO_PIN_12, GPIO_PIN_RESET );
HAL_Delay( 1 );
```

W kolejnej fazie następuje zerowanie licznika 14. Ustawienie portu PA12 powoduje ładowanie pojemności sondy:

```
//----- ładowanie pojemności
TIM14->CNT = 0;
HAL_GPIO_WritePin( GPIOA, GPIO_PIN_12, GPIO_PIN_SET );
HAL_Delay( 1 );
```

Wzrastające napięcie w czasie ładowania wywołuje przerwanie na wejściu PC14, w którego obsłudze wywoływana jest funkcja „irqWejsciePomiarowe”:

```
void EXTI4_15_IRQHandler(void)
{
  /* USER CODE BEGIN EXTI4_15_IRQn 0 */

      extern void irqWejsciePomiarowe(void);
       irqWejsciePomiarowe();

  /* USER CODE END EXTI4_15_IRQn 0 */
  HAL_GPIO_EXTI_IRQHandler(GPIO_PIN_7);
  /* USER CODE BEGIN EXTI4_15_IRQn 1 */

  /* USER CODE END EXTI4_15_IRQn 1 */
}
```

W handlerze obsługi przerwania wejść 4...15 nie jest określane wejście, które wywołało przerwanie, a to dlatego, że tylko wejście 14 ma włączone przerwania. W funkcji „irqWejsciePomiarowe” odczytywany jest czas zliczony przez timer14:

```
void irqWejsciePomiarowe(){
  uint8_t static nrPomiaru = LICZBA_POMIAROW;


  CzasLadowania = TIM14->CNT;
  uint32_t static srednia;
  srednia += CzasLadowania;
  if( ! nrPomiaru ){
    SredniaLadowania = srednia / LICZBA_POMIAROW;
     srednia = 0;
  }
  nrPomiaru--;
  nrPomiaru &= LICZBA_POMIAROW-1;
}
```

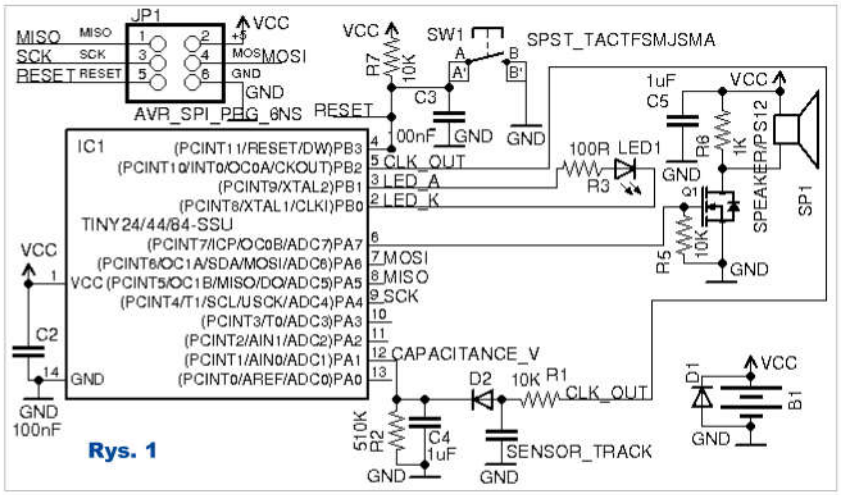
Wyniki kilku pomiarów są uśredniane i zapisywane w zmiennej „SredniaLadowania”, która jest porównywana z wartością alarmową, której przekroczenie powoduje odtworzenie samplowanego dźwięku. Sample odtwarzane są przez PWM kanał 3 timera 3, którego konfigurację przedstawia **rysunek 2**. Handler przerwania timera 3 określa przyczynę przerwania, jeśli jest to przepełnienie licznika, wywoływana jest funkcja „irqPlaySample”:

```
void TIM3_IRQHandler(void)
{
  /* USER CODE BEGIN TIM3_IRQn 0 */

   extern TIM_HandleTypeDef htim3;
  if (__HAL_TIM_GET_FLAG(&htim3, TIM_FLAG_UPDATE) != RESET)
```

Tabela 2

	Tiny85	STM32G0
	---	---
Rdzeń	8-bit	32-bit
Zegar	20MHz	64MHz
FLASH	8kB	32kB
RAM	512B	8kB
EEPROM	512B	- *
Timery	2 (8 i 16 bit)	8 (16-bit) + SYS w rdzeniu
UART	-	2 (wykrywanie prędkości i końca ramki, sprzętowe sterowanie przepływem, sprzętowe sterowanie kierunkiem RS485)
I2C	USI **	2 (1MHz)
SPI	USI (10Mb/s) **	2 (32 Mb/s)
RTC	-	+
DMA	-	1 (5 kanałów)
ADC10-bit	10-bit	12-bit
WDG	+	2 (tradycyjny i okienkowy)
Generator CRC	-	+
Liczba poziomów przerwań	1	4
Kontroler przerwań	-	+
Interfejs IR	-	+

* - STM32 ma tak duży FLASH, że można w nim emulować EEPROM. Przeważnie w mikrokontrolerach EEPROM jest emulowany w pamięci FLASH „poza świadomością programisty”.
** - Albo I2C albo SPI. Oba interfejsy z silnym wspomaganiem programowym. Realny transfer SPI ok. 5Mb/s.



Rys. 1

```
    {
      if (__HAL_TIM_GET_IT_SOURCE(&htim3, TIM_IT_UPDATE) != RESET)
       {
          extern void irqPlaySample(void);
          irqPlaySample();
       }
    }


  /* USER CODE END TIM3_IRQn 0 */
  HAL_TIM_IRQHandler(&htim3);
  /* USER CODE BEGIN TIM3_IRQn 1 */

  /* USER CODE END TIM3_IRQn 1 */
}
```

W funkcji przerwania zapisywane są kolejne próbki do PWM:

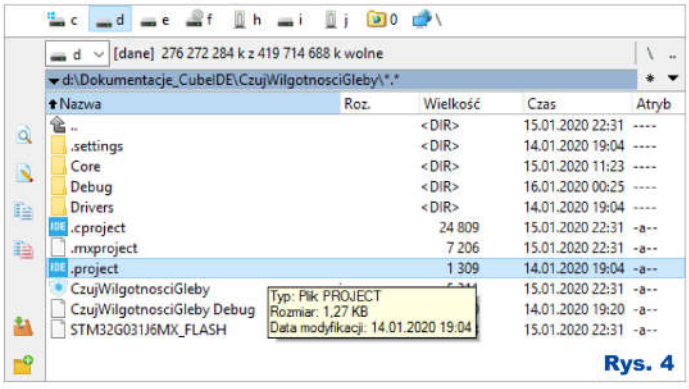
```
void irqPlaySample(){
  uint8_t static div;
  if( --div == 0 ) {
    div = 4;     // 16MHz / 4 / 2 = 7`812,5Hz

    uint8_t pwm = *pSample++;
    if( pwm & 0x80 ){     // Konwersja U8 na I8
      pwm = ~pwm;
      pwm++;
    }
    __HAL_TIM_SET_COMPARE(&htim3, TIM_CHANNEL_3, pwm );

    if( --lenSample == 0 ) stanSample=0;
  }
}
```

Próbka przed zapisaniem jest konwertowana ze zmiennej ze znakiem na zmienną bez znaku, dlatego nie można było skorzystać z mechanizmu DMA. Tak naprawdę co czwarte przerwanie następuje zapis kolejnej próbki do PWM, co pozwala uzyskać lepszą jakość dźwięku.

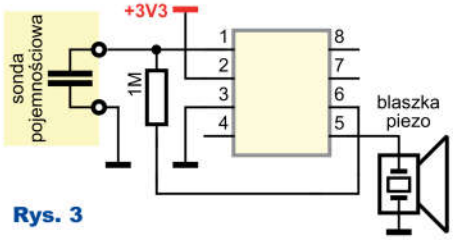
Czytelników, którzy chcieliby do pomiaru użyć Arduino i funkcji „PulseIn", muszę rozczarować. Funkcja ta napisana jest źle (jak większość wszystkiego co dotyczy Arduino, nawet DigitalWrite ma poważne błędy). W konsekwencji pomiar krótkich czasów będzie obarczony dużymi błędami. Co je powoduje? Gdy przyjrzymy się funkcji:

```
unsigned long pulseIn(uint8_t pin, uint8_t state, unsigned long timeout)
{
  uint8_t bit = digitalPinToBitMask(pin);
  uint8_t port = digitalPinToPort(pin);
  uint8_t stateMask = (state ? bit : 0);

  unsigned long maxloops = microsecondsToClockCycles(timeout)/16;

  unsigned long width = countPulseASM(portInputRegister(port), bit, stateMask, maxloops);

  if (width)
    return clockCyclesToMicroseconds(width * 16 + 16);
  else
    return 0;
}
```

oraz asemblerowej wstawce „countPulseASM" w pliku „wiring_pulse.S" (szkoda stron czasopisma na publikowanie tej źle napisanej funkcji, można ją znaleźć w materiałach dodatkowych), to zauważymy, że pomiar czasu realizowany jest programowo, a przerwania nie są zwieszane (globalne zawieszanie przerwań to zła praktyka stosowana między innymi w przypadku obsługi 1-Wire czy WS2812 w sytuacji, gdy można je obsłużyć bez zawieszania przerwań czy w przerwaniach), co w konsekwencji prowadzi do błędów, gdy w czasie pomiaru obsłużone zostanie przerwanie, a są obsługiwane przerwania od timera 0. Skorzystanie z wątpliwej jakości bibliotek obsługujących UART czy I2C używających przerwań wprowadzi kolejne błędy pomiarowe. Wydawać się może, że przed „PulseIn" wystarczy zablokować przerwania, a po jej wykonaniu odblokować, ale globalne zawieszanie przerwań jest złą praktyką. Dobrze napisany program nie zawiesza globalnych przerwań, a biblioteki Arduino czynią to nader często (przeważnie zadanie można zrealizować bez zawieszania przerwań), z czego wynikają problemy, gdy próbuje połączyć się kilka funkcjonalności w jedną całość.



Rys. 4



Rys. 3

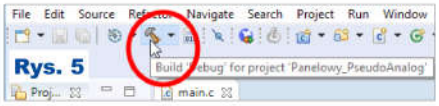
Powstaje wtedy potworek bardzo podobny do Windows, który przez 25 lat nie rozwiązał problemów, które inne systemy rozwiązały ponad 30 lat wcześniej.

Przyznam, że nie znam Arduino na wylot, bo to, co poznałem, skutecznie mnie do Arduino zniechęciło. Nie znalazłem jeszcze poprawnie napisanej biblioteki dla Arduino, ale jeśli w banalnym DigitalWrite są błędy, to czemu na siłę mam szukać jednej dobrej biblioteki spośród tysięcy złych?

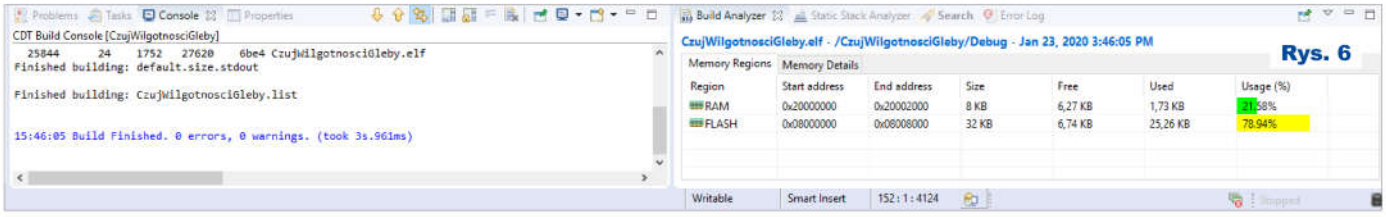
Montaż i uruchomienie

Układ został zmontowany „na pająka" według schematu z **rysunku 3** (przerysować). Prototyp został zbudowany z wykorzystaniem zestawu startowego STM32G0316-DISCO, zawierającego, tradycyjne dla zestawów STM, programator/debugger ST-LINK. Zestaw można kupić za ok. 50zł, sam mikrokontroler za ok. 3zł. **Fotografie w artykule** pokazują model.

Zmontowany układ należy uruchomić. Potrzebne do tego będzie zainstalowanie środowiska CubeIDE, które można pobrać ze strony producenta mikrokontrolerów STM32 *https://www.st.com/en/development-tools/stm32cubeide.html#tools-software*. Gdy środowisko jest zainstalowane i pobierze wymagane pliki z Internetu, można otworzyć projekt, klikając w ikonę projektu – **rysunek 4**. Kod kompilujemy, naciskając symbol młotka – **rysunek 5**. Raport z kompilacji ukazuje się w lewym oknie na dole, po prawej znajdziemy informacje o zajętości pamięci mikrokontrolera – **rysunek 6**. Program wgrywamy do mikrokontrolera, naciskając symbol robaka – **rysunek 7**. Zanim to zrobimy, należy podłączyć

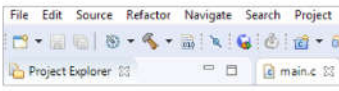


Rys. 5

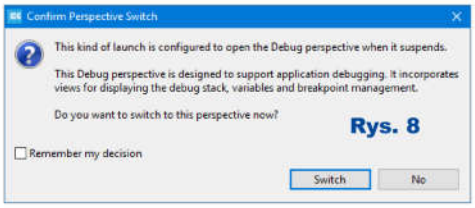


Rys. 6

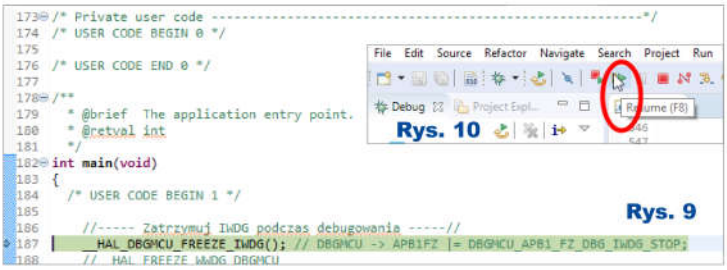
programator do mikrokontrolera, naturalnie gdy używamy zestawu DISCOVERY, połączenie jest już wykonane. Linię SWD CK programatora przyłączamy do wyprowadzenia 8 mikrokontrolera, SWD IO do wyprowadzenia 7. Po wgraniu programu do mikrokontrolera pojawi się okno informujące o przełączeniu widoku (**rysunek 8**) – naciskamy „Switch". Można także, używając klawisza F11, program zarówno skompilować jak i po poprawnej kompilacji wgrać do mikrokontrolera. Program zatrzyma się na pierwszej linii programu – **rysunek 9**. Aby go uruchomić, należy nacisnąć przycisk „PLAY" (**rysunek 10**) lub klawisz F8.



Rys. 7

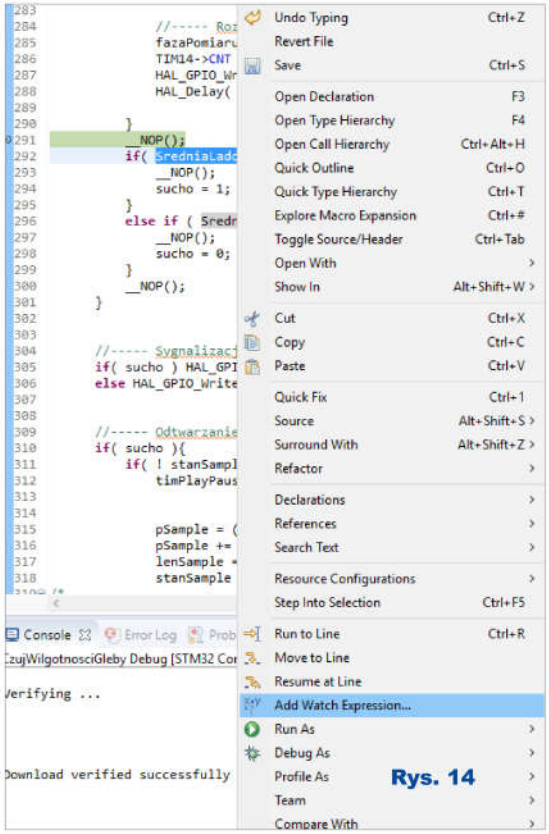


Rys. 8

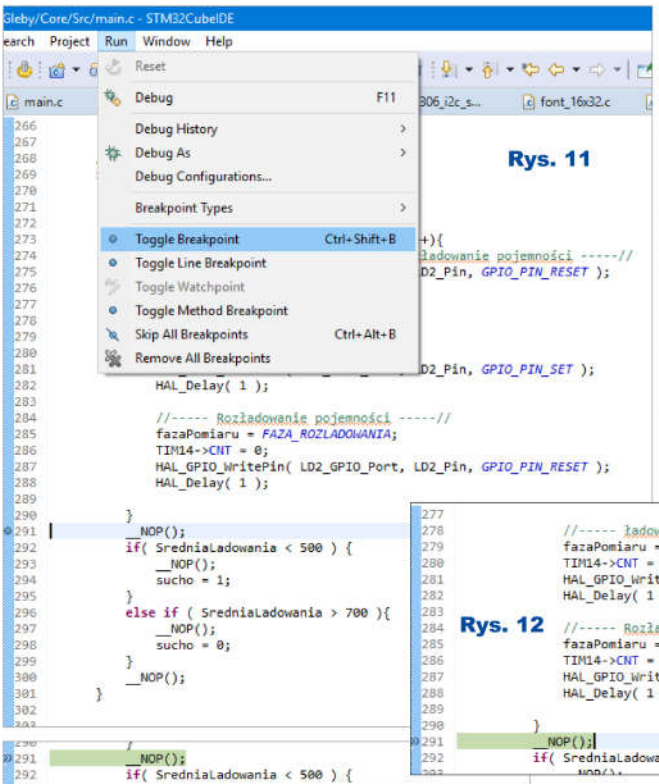


Rys. 10
Rys. 9

Aby ustawić próg wilgotności uznawany za suchą glebę, należy w niej umieścić sondę. W programie w linii 291 trzeba ustawić pułapkę. Można zrobić to na kilka sposobów. Najwygodniejszy to kliknąć dwa razy na niebieskim pasku przed numerem linii. Można też wybrać opcje z menu – **rysunek 11** lub nacisnąć kombinację CTRL+SHIFT+B. Po uruchomieniu (F8) program zatrzyma się na pułapce – **rysunek 12**. Wystarczy sprawdzić wartość zmiennej „SredniaLadowania", aby wiedzieć, jaka wartość zliczona przez licznik odpowiada glebie suchej. Jak odczytać wartość zmiennej? Wystarczy umieścić na niej kursor i chwilę zaczekać, aby zobaczyć okno z informacjami o zmiennej – **rysunek 13**. Takie podglądanie wielu zmiennych byłoby niewygodne, dlatego można umieścić podgląd w okienku zakładki „Expressions" z prawej strony. Aby to zrobić, najprościej zaznaczyć zmienną, po czym nacisnąć prawy klawisz myszki i wybrać z menu „Add Watch Expression..." – **rysunek 14**. W oknie, które się pojawi, naciskamy OK – **rysunek 15**. Od tej chwili zmienna jest widoczna w okienku – **rysunek 16**. Po każdym zatrzymaniu programu ujrzymy nowe wartości zmiennych. Istnieje możliwość obserwowania

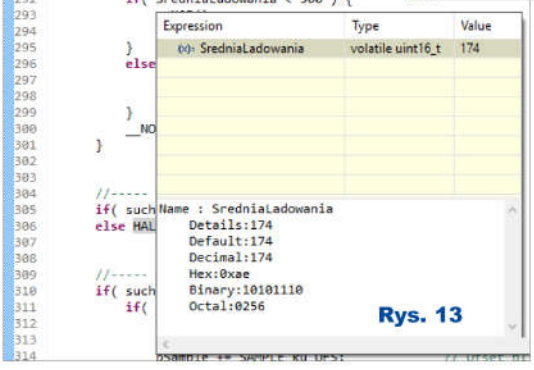


Rys. 14

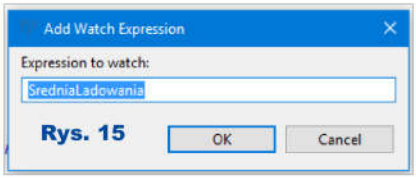


Rys. 11

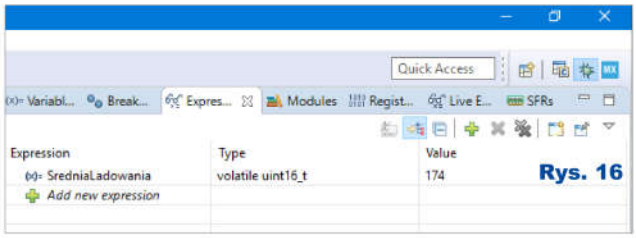
Rys. 12



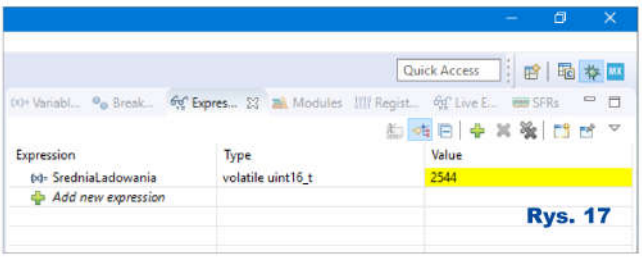
Rys. 13



Rys. 15



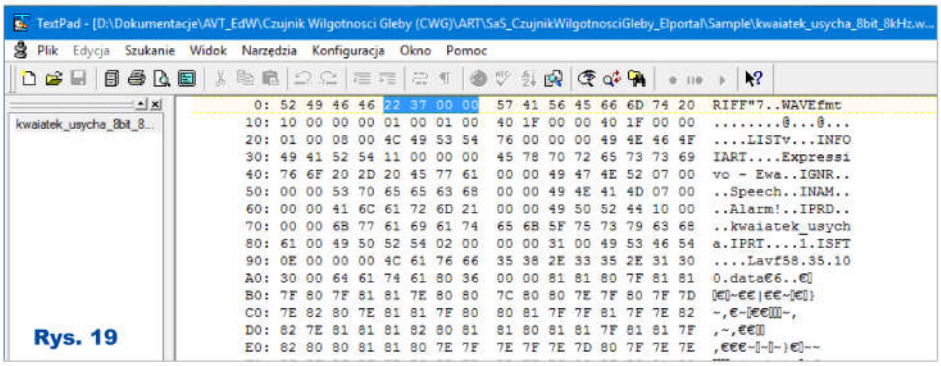
Rys. 16



Rys. 17

wartości zmiennych on-line, ale to temat na inną okazję. Zmienne, które zmieniły wartość, wyświetlane są na żółtym tle – **rysunek 17**. Znając wartość gleby suchej, można zmodyfikować wartości warunków, które decydują, kiedy gleba jest sucha, a kiedy mokra:

```
if( SredniaLadowania < 500 ) {
  sucho = 1;
}
else if ( SredniaLadowania > 700 ){
  sucho = 0;
}
```



Przygotowanie sampli. Plik dźwiękowy należy skonwertować do standardu WAV 8-bit 8kHz. Do tego celu użyto Audio Online Convertera dostępnego pod linkiem *https://audio.online-convert.com/*. Ustawienia konwersji prezentuje **rysunek 18**. Plik WAV należy skonwertować do postaci kodu źródłowego w C. W tym celu można użyć programu „bin2c.exe" dostępnego w materiałach dodatkowych. Wygenerowany plik należy umieścić w katalogu projektu i dołączyć dyrektywą „include" do programu:

```
#include "../Sample/kwiatek_usycha_8bit_8kHz.h"
```

Trzeba jeszcze ustawić kilka definicji:

```
#define  SAMPLE_ku_OFS    0xAA
#define  SAMPLE_ku_DATA   kwiatek_usycha_8bit_8kHz_wav
#define  SAMPLE_ku_LEN    (kwiatek_usycha_8bit_8kHz_wav_size - SAMPLE_ku_OFS)
```

SAMPLE_ku_OFS to ofset wskazujący, gdzie zaczynają się dane sampli. Program mógłby te dane odczytać z nagłówka pliku, ale ze względu na to, że to program demonstracyjny, zdecydowano się na ręczne ustawienie ofsetu, a przy okazji można poznać strukturę pliku WAV. Plik składa się z CHUNK-ów. Pierwszy CHUNK tworzy ciąg „RIFF", kolejne cztery bajty to rozmiar pliku – **rysunek 19**. Kolejny CHUNK zaczyna się zaraz za pierwszym, tworzy ciąg „WAVEfmt" i ma rozmiar 16 bajtów (czerwone podkreślenie na **rysunku 20**). Nie będę opisywał struktury CHUNK-a, w której zawarte są informacje o częstotliwości samplowania, liczbie kanałów, wielkości próbek itp., bo wiemy, w jakim formacie plik został zapisany. Od adresu 0x24 znajduje się kolejny CHUNK „LIST" – **rysunek 21**. Nie jest to standardowy CHUNK, jest on dodawany przez program „Expressivo". Jego wielkość to 0x76 bajtów. Aby wyliczyć adres kolejnego CHUNK-a, do adresu bieżącego dodajemy jego rozmiar + 8 bajtów (osiem bajtów to nazwa CHUNK-a i dane o jego rozmiarze). Wynik działania 0×24 + 0×76 + 8 = A2. Pod adresem 0×A2 znajduje się CHUNK „data" – **rysunek 22**. Jego rozmiar to 0x3683 czyli 13952 bajtów. Od adresu 0xAA zaczynają się dane sampli i taką wartość należy przypisać do *SAMPLE_ku_OFS*. Można zaoszczędzić kilkadziesiąt bajtów FLASH, w tym przypadku 170 (0xAA), ustalając offset na 0 i jednocześnie wycinając z dołączanego pliku owe 170 bajtów.

Instalację i obsługę CubeIDE opisałem bardzo pobieżnie. Wynika to z tego, że nie da się w kilku zdaniach opisać nawet podstawowej obsługi tak potężnego narzędzia. Na ten cel trzeba by przeznaczyć kilkanaście stron czasopisma. Jeśli Czytelnicy są zainteresowani takim artykułem, proszę o e-maile do redakcji.

SaS
sas@elportal.pl