Zastosowania procesorów sygnałowych

WPROWADZENIE DO PROGRAMOWANIA PROCESORÓW SYGNAŁOWYCH

na przykładzie procesora C5535

Opracowanie: Grzegorz Szwoch

Politechnika Gdańska, Katedra Systemów Multimedialnych

Wprowadzenie

W jaki sposób tworzymy oprogramowanie podczas laboratorium ZPS?

- Układ uruchomieniowy DSP C5535 firmy Texas Instruments.
- Procesor stałoprzecinkowy (nie ma *float* / *double*!).
- Używamy programu (IDE) *Code Composer Studio*.
- Piszemy kod programu w języku C.
- Zmiany w kodzie źródłowym wymagają ponownego skompilowania (zbudowania) programu. Podczas laboratorium budujemy kod w trybie *Debug*.
- Jeżeli program zbudował się bez błędów, można go uruchomić na układzie uruchomieniowym („płytce DSP”), podłączonym do USB komputera.
- Program może działać w trybie ciągłym.
- Możemy też zatrzymać program (*breakpoint*), uruchamiać program linijka po linijce (tryb krokowy), podglądać zawartość zmiennych, itp.

Ogólna struktura kodu w języku C (przykład)

```c
// Dołączenie potrzebnych plików nagłówkowych
#include "dsplib.h"

// Obszar deklaracji zmiennych globalnych
short bufor[2048];

// Własne funkcje (opcjonalne)
short kwadrat(short x) {
    return _smpy(x, x);
}

// Główna procedura - tutaj zaczyna się program
void main(void) {
    // Obszar deklaracji zmiennych lokalnych (na stosie)
    short lewy, prawy;
    // Przetwarzanie próbek sygnału w pętli
    while (1) {
        aic3204_codec_read(&lewy, &prawy);
        lewy = kwadrat(lewy);
        prawy = kwadrat(prawy);
        aic3204_codec_write(lewy, prawy);
    }
}
```

Uwagi o wskaźnikach w języku C

- Nagłówek funkcji w języku C informuje o typach argumentów i wyniku.

```
void aic3204_codec_read(short* left_input, short* right_input);
```

- Gwiazdka (*) oznacza, że wywołując funkcję musimy podać wskaźnik do argumentu.
- Aby pobrać wskaźnik do zmiennej „skalarnej” (*short*, *long*, itp.), musimy postawić znak & przed nazwą zmiennej.

```
short lewy, prawy;
aic3204_codec_read(&lewy, &prawy);
```

- Tablice deklarujemy podając [rozmiar] po nazwie zmiennej.
  Nazwa zmiennej jest już wskaźnikiem do tablicy.

```
short tablica[1024];
rfft(tablica, 2048, SCALE);      // void rfft(short* x, ushort nx, type)
```

Przygotowanie procesora do pracy

```
// Dołączenie potrzebnych plików nagłówkowych
#include "usbstk5515.h"
#include "usbstk5515_led.h"
#include "aic3204.h"
#include "PLL.h"
#include "bargraph.h"
#include "oled.h"
#include "pushbuttons.h"

void main(void) {
    // Inicjalizacja układu DSP
    USBSTK5515_init();        // przygotowanie płytki do komunikacji USB
    pll_frequency_setup(100); // częstotliwość taktowania procesora 100 MHz
    aic3204_hardware_init();  // I2C
    aic3204_init();           // kodek dźwięku AIC3204
    USBSTK5515_ULED_init();   // diody LED
    SAR_init_pushbuttons();   // przyciski (przetwornik A/C)
    oled_init();              // wyświelacz LED 2x19 znaków
    // ustawienie częstotliwości próbkowania dźwięku i wzmocnienia wejścia dźwięku
    set_sampling_frequency_and_gain(48000L, 30);
    // ... dalszy ciąg programu
}
```

Przetwarzanie próbek dźwięku w pętli

W nieskończonej pętli: odczytujemy wartości próbek (stereo) z wejścia, przetwarzamy próbki w dowolny sposób, zapisujemy przetworzone próbki na wyjście.

```
void main(void) {
    // ... Inicjalizacja układu DSP

    // zmienne do przechowywania wartości próbek
    short lewy_wejscie, prawy_wejscie, lewy_wyjscie, prawy_wyjscie;

    // przetwarzanie próbek sygnału w pętli
    while (1) {
        // odczyt próbek dźwięku
        aic3204_codec_read(&lewy_wejscie, &prawy_wejscie);

        // tutaj przetwarzamy próbki dźwięku, np.:
        lewy_wyjscie = lewy_wejscie;
        prawy_wyjscie = prawy_wejscie;

        // zapisanie wartości na wyjście dźwiękowe
        aic3204_codec_write(lewy_wyjscie, prawy_wyjscie);
    }
}
```

Tryby pracy

W szablonie kodu zdefiniowano cztery tryby pracy, które można przełączać za pomocą dwóch przycisków (do tyłu, do przodu).

W każdym trybie można zaprogramować inny sposób przetwarzania próbek.

Jeden z trybów można pozostawić jako kopiowanie wejścia („bypass”) w celu porównania z sygnałem przetworzonym.

```
if (tryb_pracy == 1) {
    lewy_wyjscie = lewy_wejscie;
    prawy_wyjscie = prawy_wejscie;
} else if (tryb_pracy == 2) {
    lewy_wyjscie = kwadrat(lewy_wejscie);
    prawy_wyjscie = kwadrat(prawy_wejscie);
} else if (tryb_pracy == 3) {
    lewy_wyjscie = 0;
    prawy_wyjscie = 0;
} else if (tryb_pracy == 4) {
    lewy_wyjscie = 0;
    prawy_wyjscie = 0;
}
```

Zmienne i stałe

- Zmienne deklarowane poza funkcjami są globalne – widoczne w całym kodzie.
- Zmienne deklarowane wewnątrz funkcji są lokalne, umieszczane są na stosie i są one widoczne tylko wewnątrz bloku kodu, w którym jest deklaracja.
- Stałe (przedrostek *const*) nie mogą mieć zmienianej wartości.

```
short lewy_wejscie;     // zmienna globalna, nie zainicjalizowana
short licznik = 0;      // zmienna globalna, zainicjalizowana
short bufor[1024];      // zmienna globalna, tablica o rozmiarze 1024 (nie zainicjalizowana)
const short ROZMIAR_BUFORA = 1024;      // stała globalna

void main(void) {
    int i; // zmienna lokalna
    for (i = 0; i < ROZMIAR_BUFORA; ++i) {
        bufor[i] = 0; // odwołanie do zmiennej globalnej
    }
}
```

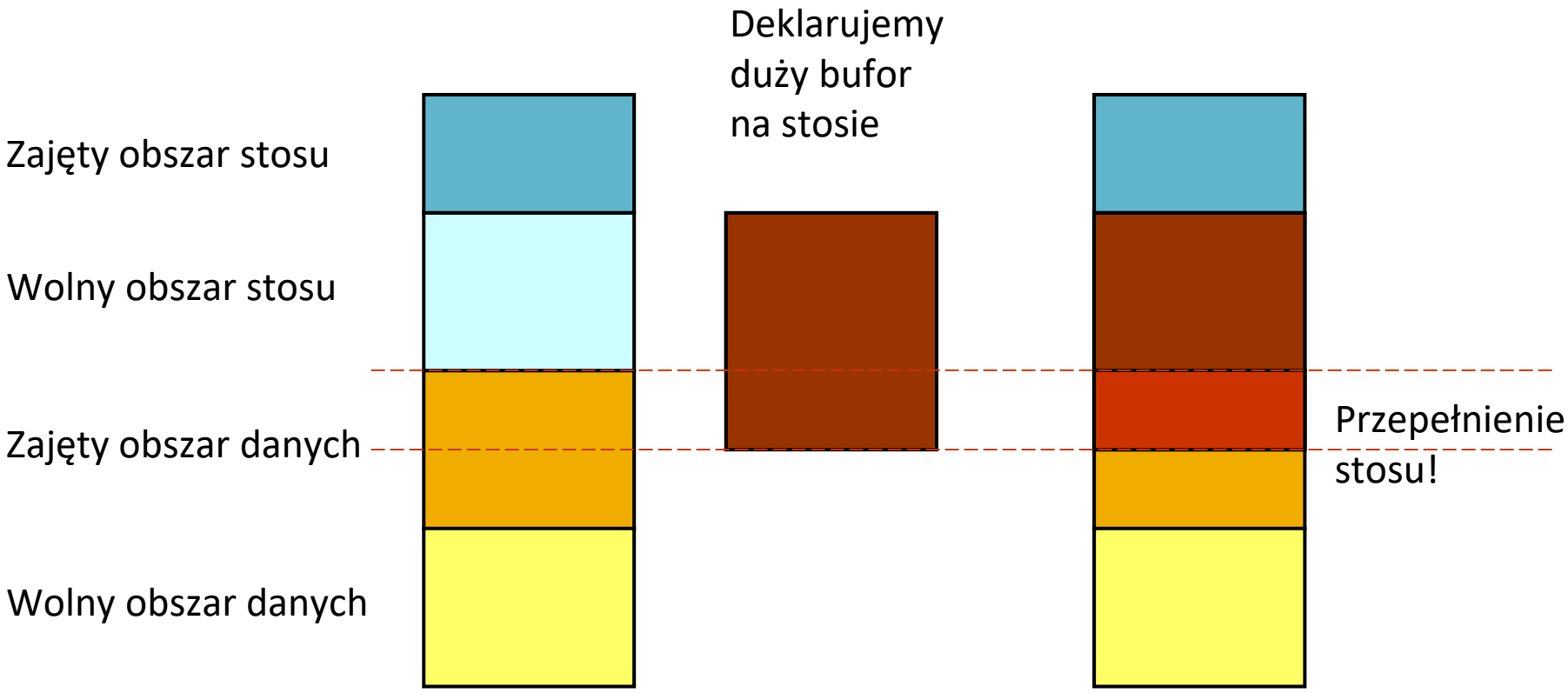
Stos

- Wszystkie zmienne deklarowane wewnątrz funkcji są umieszczane są na stosie.
- Stos ma zwykle ograniczony rozmiar. Na naszym procesorze: domyślnie 4 KB.
- Jeżeli zmienna nie zmieści się na stosie, następuje przepełnienie stosu (*stack overflow*). Dane znajdujące się za stosem zostają nadpisane.
- W naszym przypadku nie dostajemy żadnej informacji o tym!
- Program może działać poprawnie, może generować błędne wyniki albo może się zawiesić. Znalezienie przyczyny problemu jest często trudne.
- Wniosek: nie deklarujemy dużych tablic na stosie.

```
short bufor[1024];      // OK - tablica jako zmienna globalna, umieszczana w pamięci ogólnej

void main(void) {
    short bufor2[1024];      // ŹLE - tutaj próbujemy utworzyć tablicę na stosie,
                             // może to spowodować przepełnienie stosu
}
```

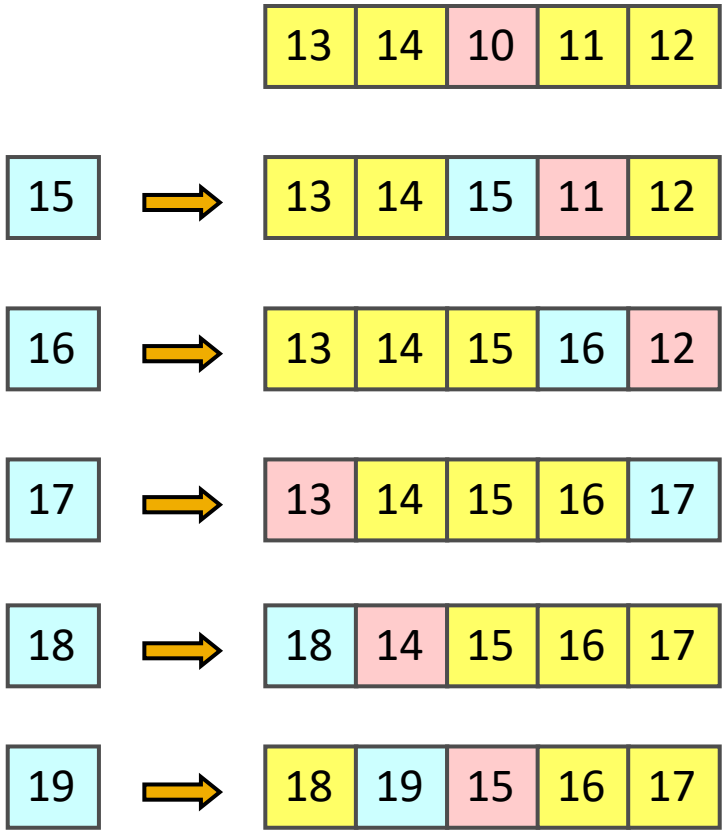
Przepełnienie stosu

Ilustracja przepełnienia stosu przez próbę deklaracji dużej tablicy wewnątrz funkcji.



Zapisywanie wartości do tablicy

- Załóżmy, że potrzebujemy mieć zawsze dostępnych *N* ostatnich wartości (np. do filtracji).
- Tworzymy tablicę o długości N.
- Utrzymujemy wskaźnik zapisu – zmienną, która wskazuje na pozycję „najstarszej wartości”. Komórka pod wskaźnikiem zostanie nadpisana nową wartością.
- Po zapisaniu komórki, przesuwamy wskaźnik na kolejną pozycję.
- Jeżeli wskaźnik przejdzie poza koniec tablicy – zawijamy go (wraca na zero).
- W ten sposób tworzymy bufor kołowy (*circular buffer*).



Bufor kołowy w praktyce

```
#define DLBUF 1024   // Dobrym pomysłem jest zapisanie długości bufora jako tzw. stałej kompilacji
                     // UWAGA: nie ma średnika po instrukcji #define (!!!)
short bufor[DLBUF];  // globalna deklaracja bufora kołowego
short indeks = 0;    // wskaźnik pozycji zapisu do bufora kołowego

void main(void) {
    // ... Inicjalizacja układu DSP.
    // ... Tutaj przetwarzamy próbki sygnału, obliczamy zmienną lewy_wyjscie.

    // zapis wartości do bufora kołowego
    bufor[indeks] = lewy_wejscie;
    // przesunięcie wskaźnika
    indeks++; // to samo co: indeks = indeks + 1, albo: indeks += 1
              // można połączyć dwie instrukcje: bufor[indeks++] = lewy_wyjscie;
    // sprawdzamy czy jestesmy na koncu tablicy
    if (indeks == DLBUF) {    // ostatnia pozycja w tablicy to DLBUF-1
        indeks = 0; // zawijamy wskaźnik na 0
        // w tym miejscu wiemy, że zapisaliśmy całą tablicę nowymi próbkami
    }
}
```

Adresowanie bufora kołowego

Jeżeli nie musimy sprawdzać czy wskaźnik doszedł do końca bufora kołowego, to możemy użyć specjalnej instrukcji wewnętrznej procesora C5535 do adresowania kołowego: *_circ_incr*. Jest ona szybsza niż „ręczne” przesuwanie wskaźnika.

```
void main(void) {
    // ...
    // zapis wartości do bufora kołowego
    bufor[indeks] = lewy_wejscie;
    // przesunięcie wskaźnika o 1, z automatycznym zawinięciem
    indeks = _circ_incr(indeks, 1, DLBUF);
}
```

Obliczanie iloczynu skalarnego

Mamy *N* ostatnich wartości zapisanych w buforze kołowym. Teraz musimy przemnożyć te wartości (od najnowszej do najstarszej) przez współczynniki (zapisane w tablicy) i zsumować wyniki mnożenia.

```
// const short wspolczynnik[DLBUF] = { ... };  // tablica współczynników

void main(void) {
    // ...
    bufor[indeks] = lewy_wejscie;
    short n = indeks; // zaczynamy od najnowszej wartości w buforze
    long wynik = 0;   // wynik operacji
    short i;
    for (i = 0; i < DLBUF; ++i) {
        wynik += _smpy(bufor[n], wspolczynnik[i]); // mnożenie i dodawanie (dwie instrukcje)
        n = _circ_incr(n, -1, DLBUF); // przesunięcie wskaźnika o 1 do tyłu, z zawinięciem
    }
    indeks = _circ_incr(indeks, 1, DLBUF); // dopiero tutaj przesuwamy główny wskaźnik
}
```

Obliczanie iloczynu skalarnego - MAC

Dla każdej wartości trzeba wykonać mnożenie i dodawanie jako osobne instrukcje procesora (2 cykle).

Zamiast tego można użyć instrukcji MAC (*multiply and accumulate*), która wykonuje obie operacje w jednym cyklu: $y := y + a \cdot b$.

Mamy do tego instrukcję wewnętrzną: *_smac*.

```
void main(void) {
    // ...
    bufor[indeks] = lewy_wejscie;
    short n = indeks; // zaczynamy od najnowszej wartości w buforze
    long wynik = 0;   // wynik operacji
    short i;
    for (i = 0; i < DLBUF; ++i) {
        wynik = _smac(wynik, bufor[n], wspolczynnik[i]); // mnożenie i dodawanie jako MAC
        n = _circ_incr(n, -1, DLBUF); // przesunięcie wskaźnika o 1 do tyłu, z zawinięciem
    }
    indeks = _circ_incr(indeks, 1, DLBUF);
}
```

DSPLIB

- Dla procesora sygnałowego C5535 firma Texas Instruments dostarcza bibliotekę *DSPLIB*. Zawiera ona typowe operacje przetwarzania sygnałów, napisane w asemblerze i zoptymalizowane pod ten procesor.
- Zawiera: FFT (długość musi być potęgą dwójki do 2048), filtry (FIR i IIR) i inne.
- Warto korzystać z tych procedur zamiast pisać własne.
- Jeżeli potrafimy programować w asemblerze, możemy je dostosować do naszych potrzeb (kody źródłowe są dostępne).
- Wywołujemy procedury *DSPLIB* z naszego kodu jako funkcje.
- Wymagane jest dołączenie nagłówka: `#include <dsplib.h>`.
- Dokumentacja: https://www.ti.com/lit/ug/spru422j/spru422j.pdf

DSPLIB – obliczanie FFT

Przykład: obliczanie FFT. Ustalamy rozmiar transformaty: 1024.

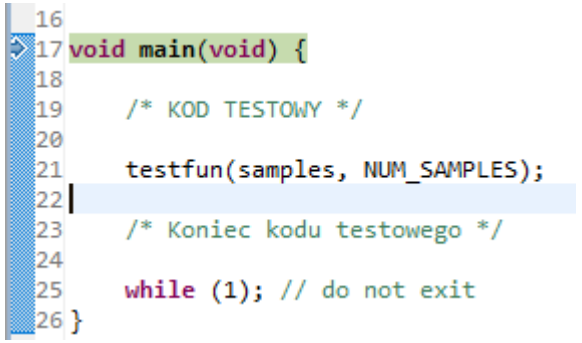
Zapisujemy 1024 wartości w buforze, po jego zapełnieniu: wywołujemy funkcję.

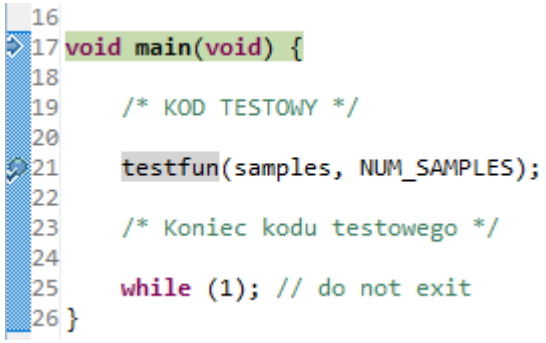
Nagłówek funkcji z dokumentacji: `void rfft (DATA *x, ushort nx, type);`

```
#define DLBUF 1024
DATA bufor[DLBUF];   // globalna deklaracja bufora kołowego (DATA == short)
short indeks = 0;    // wskaźnik pozycji zapisu do bufora kołowego

void main(void) {
    // ... Inicjalizacja układu DSP.
    // ... Tutaj przetwarzamy próbki sygnału, obliczamy zmienną lewy_wyjscie.

    bufor[indeks++] = lewy_wejscie; // zapis wartości do bufora kołowego, zwiększenie wskaźnika
    if (indeks == DLBUF) {
        indeks = 0;                 // zawijamy wskaźnik na 0
        rfft(bufor, DLBUF, SCALE);  // mamy cały bufor, więc przetwarzamy
    }
}
```

Uruchamianie programu na C5535

- Mamy program poprawnie zbudowany w *Code Composer Studio*.
- Uruchamiamy debugowanie za pomocą przycisku .
- Program uruchamia się i zatrzymuje się w punkcie startowym – na początku funkcji *main*().
- Wciśnięcie przycisku (F8) wznawia działanie programu.
- Można zrobić pauzę ( ) lub całkowicie zatrzymać program ( ).
- Dwukrotne kliknięcie na niebieskim marginesie po lewej włącza i wyłącza punkt przerwania (*breakpoint*) – program zatrzyma się na tej linii.



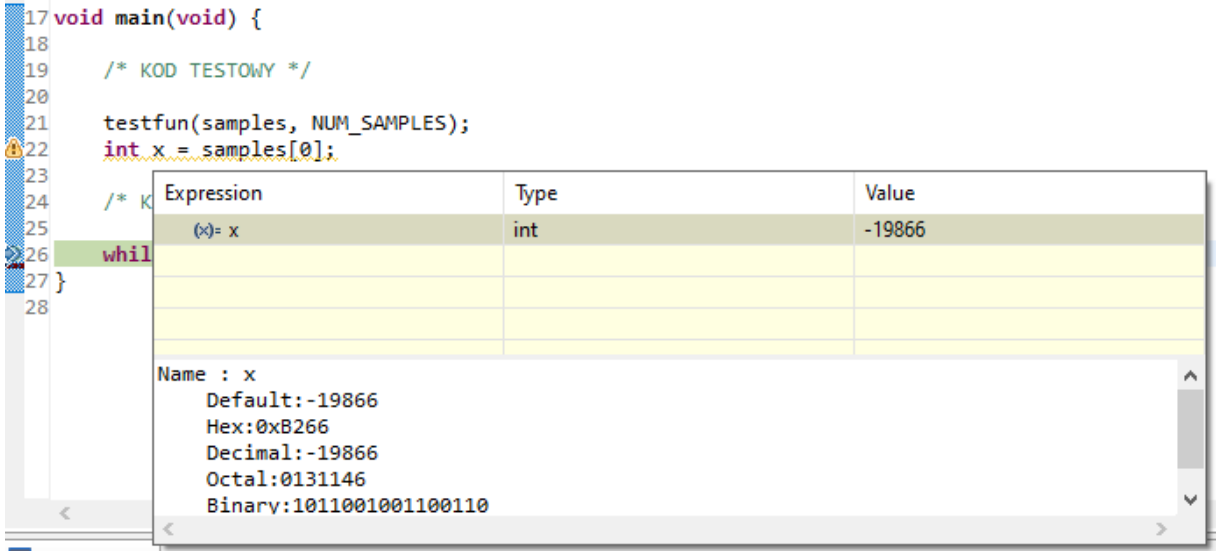


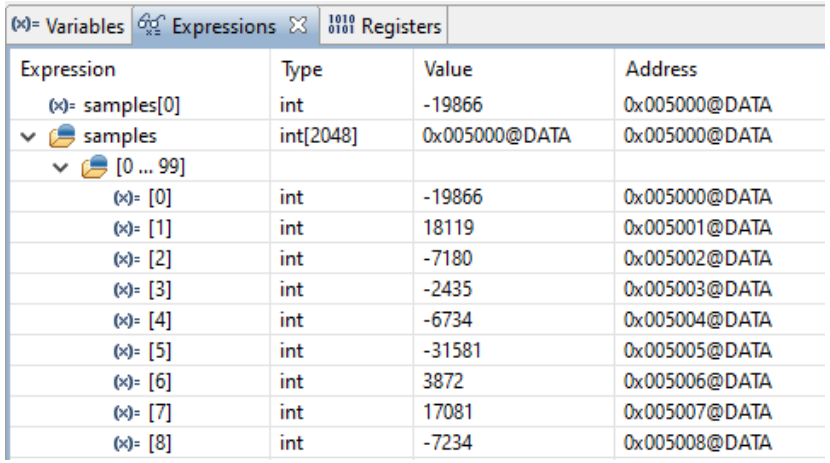
Uruchamianie programu w trybie krokowym

- Po zatrzymaniu programu możemy przechodzić po nim w trybie krokowym, linia po linii, podejrzeć zawartość zmiennych, sprawdzić poprawność i znaleźć błędy.
- (*Step Into*, F5): przejście do kolejnej instrukcji. Jeżeli instrukcją jest wywołanie funkcji, wchodzimy do wnętrza tej funkcji.
- (*Step Over*, F6): przejście do kolejnej instrukcji. Jeżeli instrukcją jest wywołanie funkcji, „przeskakujemy” tę funkcję (wykona się ona w całości).
- (*Step Return*, F7): jeżeli weszliśmy do wnętrza funkcji komendą *Step Into*, ta komenda wykonuje całą funkcję i powraca „poziom wyżej”.
- (*Restart*): uruchamia program od początku.

Podgląd wartości zmiennych

- Możemy sprawdzić wartości zmiennych po zatrzymaniu programu (*breakpoint*).
- Najechanie kursorem myszy na zmienną w kodzie pokazuje jej wartość.
- Okno *Variables* pokazuje wartości lokalnych zmiennych.
- Okno *Expressions* pozwala dodać dowolne zmienne do podglądu.



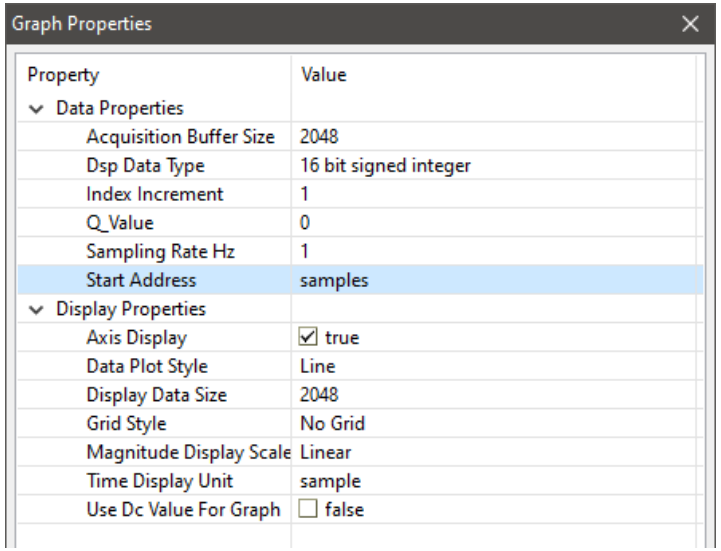


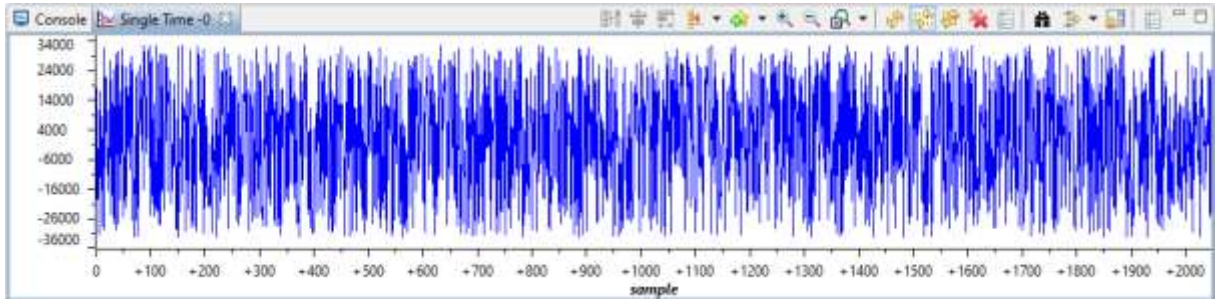
Wykreślanie zawartości tablicy

Zawartość tablicy można przedstawić za pomocą wykresu czasowego lub widma.

Służy do tego opcja *Graph* z menu *Tools*.

Szczegółowy opis: w instrukcji do laboratorium.





Podsumowanie – tworzenie programu na PS

- Projektujemy algorytm.
- Piszemy kod programu i kompilujemy go.
  - Jeżeli kompilacja się nie powiodła – mamy błędy składniowe, poprawiamy je.
  - Jeżeli się powiodła – program jest poprawny składniowo i gotowy do pracy.
- Uruchamiamy program w trybie debugowania.
  - Zatrzymujemy program, sprawdzamy wyniki.
  - Jeżeli wyniki są niepoprawne – program ma błędy logiczne. Analizujemy kod, przechodzimy przez program w trybie krokowym, podglądamy zmienne, itp. Szukamy błędów i poprawiamy je.
  - Jeżeli wyniki są poprawne – zapisujemy je.
  - Dobrym pomysłem jest testowanie programu po napisaniu samodzielnego fragmentu kodu zamiast testowania dopiero po napisaniu całości kodu.