Laboratorium PTM

Programowanie w środowisku Microchip Studio – pierwszy program w języku C.

Celem ćwiczenia jest zaznajomienie ze środowiskiem i ilustracja najprostszych programów w języku C.

Zakład Systemów Informacyjno-Pomiarowych

IETiSIP, Wydział Elektryczny, PW

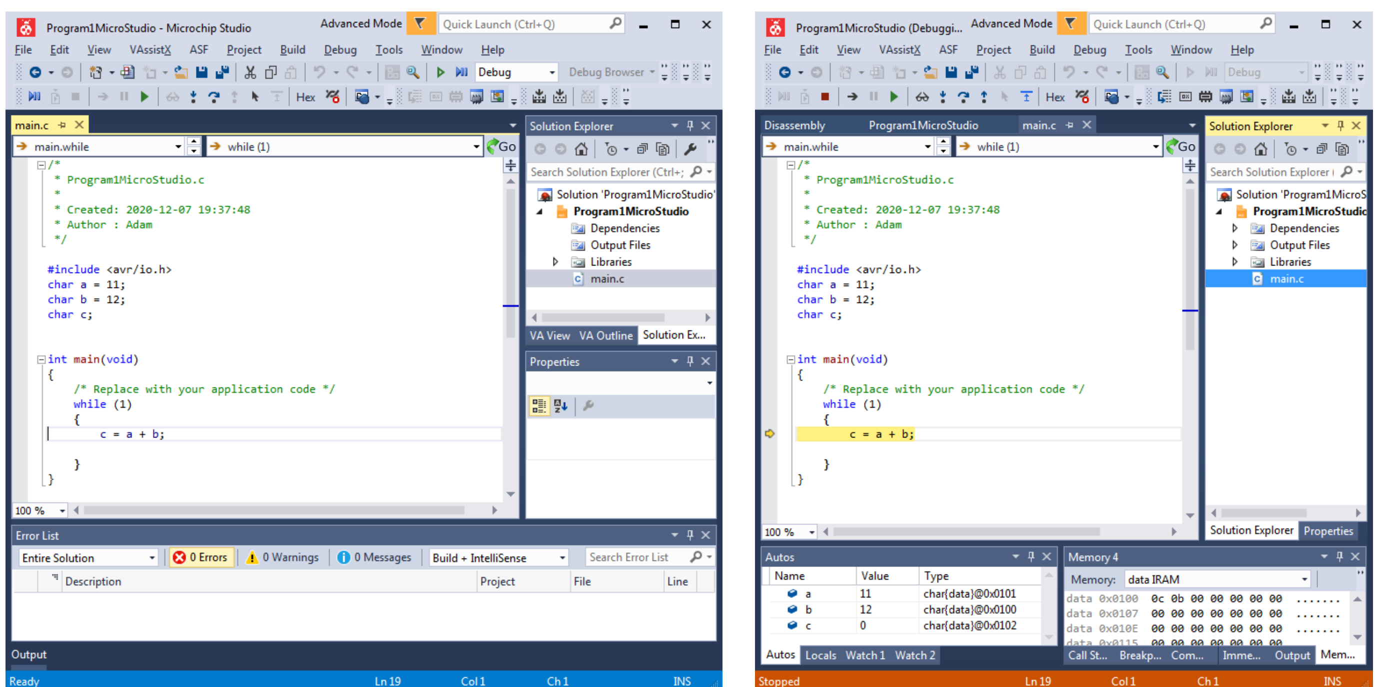
Program pierwszy w języku C:

Przy analizie kodów asemblerowych przydatna jest dokumentacja mikrokontrolera ATmega328P z listą rozkazów (i przypisanymi cyklami).

Pierwszy program w języku C dla mikrokontrolera ATmega328p niech ma postać taką jak na rysunku 1. Jak widać struktura programu jest bardzo prosta. Zawiera bowiem pojedyncze sumowanie dwóch wartości całkowitoliczbowych **a** i **b**. Wynik przechowywany jest również w zmiennej całkowitoliczbowej **c**. Wszystkie zmienne są typu **char** to znaczy, że zajmują w pamięci pojedynczy bajt i mogą przyjmować 256 wartości. Program rozpoczyna się włączeniem do programu nagłówka io.h:

```
#include <avr/io.h>
```

Nagłówek ten zawiera informacje o adresach rejestrów i wyprowadzeń używanego mikrokontrolera. Podczas tworzenia projektu wybrany został konkretny mikrokontroler a włączony nagłówek jest tu odpowiedzialny za przypisanie właściwych danych i dodatkowo zdefiniowanie nazw (literałów znakowych) dla rejestrów i wyprowadzeń konkretnego mikrokontrolera. Dzięki temu nie trzeba się szukać w dokumentacji specyficznych danych dla wybranego układu i co więcej można wykorzystywać ustandaryzowane nazwy dla poszczególnych elementów wybranego mikrokontrolera (np.: DDRB, PORTB, PINB). Zmienne w Microchip studio domyślnie są bez znaku (**unsigned**).



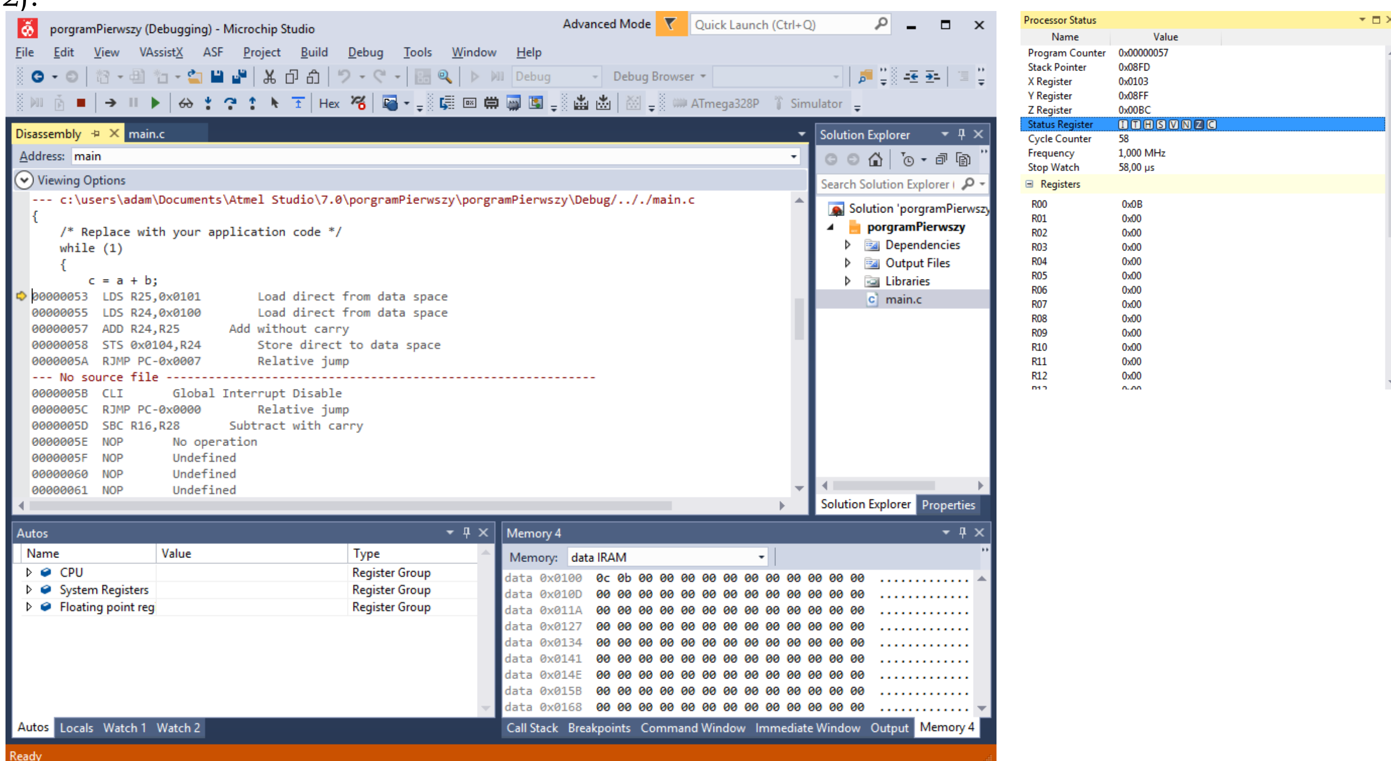
Rysunek 1. Pierwszy program w Microchip Studio.

Początkowe wartości zmiennych **a** i **b** zostały dobrane tak aby „wyróżniać się” spośród zer i jedynej w pamięci danych. To jedyny powód ich wartości. Pętla **while** pozwala na wielokrotną obserwację działania programu. Kompilację i wygenerowanie plików binarnych można wykonać poprzez Build -> **Build Solution/Application** (F7), po ewentualnych zmianach kodu można skorzystać z opcji **Rebuild** (Ctrl+Alt+F7) a „zresetowanie” projektu można uzyskać poprzez **Clean Solution/Application**. W ćwiczeniu program można uruchamiać poprzez pasek narzędziowy i **Start Debugging** (F5) oraz **Stop Debugging** (Ctrl+Shift+F5). Identycznie można sterować wykonaniem programu wybierając odpowiednie opcje z menu **Debug**. W ćwiczeniach jednak

wykorzystywać należy tryb krokowy: **Step Over** (F10). Jest to tryb pracy krokowej bez wchodzenia do funkcji/procedur (wykonywane są one w jednym kroku). Jeżeli zaistnieje potrzeba przeanalizowania działania funkcji (szczególnie samodzielnie napisanej) wtedy można użyć trybu **Step Into** (F11). Możliwość wyjścia z procedury da w tym przypadku **Step out** (Shift+F11). Powyższe opcje są standardowe dla środowisk programistycznych. Po uruchomieniu pierwszego programu (F10) okno środowiska będzie wyglądać jak na rysunku 1 (z prawej strony). Wciskając kolejno F10 program wykonywał będzie kolejne instrukcje. Żółta strzałka wskazuje aktualną instrukcję do wykonania (wiąże się to z zawartością rejestru PC – Program Counter). W tym samym obszarze, w którym znajduje się strzałka można ustawiać pułapki (breakpoints). Wystarczy użyć lewego klawisza myszy. Pułapki są użyteczne, kiedy zaistnieje potrzeba dokładnej analizy pewnego fragmentu kodu (przy założeniu, że program jest złożony niż omawiany obecnie) z pominięciem pozostałej części.

Okno **Autos** (na dole po lewej stronie) zawiera zmienne zdefiniowane w programie wraz z informacją o położeniu (lokalizacji, adresie) zmiennej w pamięci danych. Architektura mikrokontrolerów ATmega jest architekturą Harwardzką z podziałem na pamięć danych i pamięć programu. W oknie **Memory** po prawej stronie można podglądać zawartość pamięci. Przy wyborze **IRAM** (Internal RAM) można podglądać aktualną zawartość pamięci danych. Znając adresy poszczególnych zmiennych w programie (okienko **Autos** i zmienne: a, b, c) można podglądać zmiany tychże zmiennych (w takt wykonywania programu - F10). Proszę zwrócić uwagę na kolejności umieszczania zmiennych w pamięci danych.

Szczegóły wykonania programu można poznać wybierając (przy uruchomionym programie) **Debug**->**Windows**-> **Disassembly** i otrzymując w wyniku asemblerowy kod programu (rysunek 2).



Rysunek 2. Okno z kodem asemblerowym programu.

W języku asemblera również istnieje możliwość wykonywania programu w trybie krokowym. Jednakże w tym przypadku wykonywanie programu odbywa się instrukcja po instrukcji asemblerowej (NIE zaś instrukcja po instrukcji w języku C). W trakcie krokowego wykonywania programu można podglądać bieżące wartości rejestrów ogólnego przeznaczenia CPU (Rxx); rejestrów systemowych: CYCLE_COUNTER, FP, PC, SP oraz X, Y i Z oraz rejestrów wykorzystywanych przy operacjach zmiennoprzecinkowych: SREG. Wymienione rejestry dotyczą

wykorzystywanego mikrokontrolera tj. ATmega328P. Stan rejestrów można również obserwować w oknie **Memory**. Należy w tym celu wybrać podgląd na **data REGISTERS**. Adresy rejestrów można odczytać z dokumentacji mikrokontrolera. Jeszcze inną możliwością jest wybranie **Debug** -> **Windows** -> **Processor Status**, rysunek 2). W tym miejscu na rysunku 3 przestawiona została oryginalna ilustracja z mapą rejestrów mikrokontrolera ATmega328p. W przypadku podglądu zmiennych programu należy przejść do **data IRAM** (Internal RAM).

Figure 7-2. AVR CPU General Purpose Working Registers

	7 0	Addr.	
	R0	0x00	
	R1	0x01	
	R2	0x02	
	…		
	R13	0x0D	
General	R14	0x0E	
Purpose	R15	0x0F	
Working	R16	0x10	
Registers	R17	0x11	
	…		
	R26	0x1A	X-register Low Byte
	R27	0x1B	X-register High Byte
	R28	0x1C	Y-register Low Byte
	R29	0x1D	Y-register High Byte
	R30	0x1E	Z-register Low Byte
	R31	0x1F	Z-register High Byte

Rysunek 3. Rejestry ogólnego przeznaczenia mikrokontrolera ATmega328P.

UWAGA! Omawiany program został umieszczony w pamięci pod konkretnym adresem i wykorzystano w nim pewne konkretne rejestry ogólnego przeznaczenia Rxx (w programie są to rejestry R24 i R25 o adresach odpowiednio 0x18 i 0x19 – przedrostek 0x oznacza zapis heksadecymalny czyli szesnastkowy – rysunek 3). Zarówno adres początkowy programu w pamięci, jak i wykorzystane rejestry mogą się różnić w różnych kompilacjach. Dlatego podczas analizy własnych programów należy uwzględnić zarówno możliwość innego adresu początkowego zasadniczego kodu programu, inne zestawy rejestrów oraz różne położenie zmiennych w pamięci danych. Wiąże się to z koniecznością obserwacji innych obszarów pamięci.

Wartość prezentowana, jako pierwsza z lewej strony (rysunek 2) to adres w pamięci programu, czyli zawartość rejestru PC – Program Counter. Wygenerowany kod asemblerowy pochodzi z szablonu/schematu środowiska programistycznego. Fragment, który będzie w obszarze zainteresowania znajduje się poniżej komentarza z informacją o pliku źródłowym, w którym się znajduje tj. ---… /main.c. Komentarze dodane przez środowisko wyjaśniają dokładnie znaczenie poszczególnych instrukcji asemblerowych. Tak więc, kod zasadniczy pierwszego programu zaczyna się począwszy od adresu 0x0053.

```
      /* Replace with your application code */
      while (1)
      {
            c = a + b;
;Zmiennym a, b i c przypisane są adresy w pamięci danych odpowiednio: 0x101, 0x100 i
0x102.
;W programie wykorzystywane są rejestry ogólnego przeznaczenia mikrokontrolera, tu: R24
;i R25.
;Instrukcja LDS ładuje do rejestru R25 wartość danej spod adresu 0x0101 (w pamięci
;danych). W programie jest to zmienna b. Format instrukcji jest następujący:
;LDS rejestr docelowy, adres wartości źródłowej
00000053  LDS R25,0x0101      ;Load direct from data space
;Instrukcja LDS ładuje do rejestru R25 wartość danej spod adresu 0x0100 (w pamięci
;danych). W programie jest to zmienna a. Należy zwrócić uwagę na to, że ładowanie
;rejestrów wcale nie odbywa się w kolejności występowania zmiennych (a + b). Można
```

```
;sprawdzić jak zachowa się program gdy zmodyfikowana zostanie operacja dodawania na b +
;a. Format instrukcji jest następujący: LDS rejestr docelowy, adres wartości źródłowej
00000055  LDS R24,0x0100        ;Load direct from data space
;Należy zwrócić uwagę, że instrukcja ADD realizuje sumowanie wartości znajdujących się
;w rejestrach ! W tym przypadku jest to dodawanie bez przeniesienia, gdyż sumowane są
;wartości jednobajtowe - zmienne typu char. Wynik przechowywany jest w pierwszym,
;wymienionym jako argument rejestrze. W tym przypadku jest to rejestr R24.
00000057  ADD R24,R25           ;Add without carry
;Instrukcja STS przekopiowuje wynik dodawania z rejestru R24 do pamięci danych pod
;ardes 0x0102. Tam zdefiniowana jest zmienna c.
00000058  STS 0x0102,R24        ;Store direct to data space
;Ostanią instrukcją jest RJMP, która w praktyce realizuje pętlę while(1). Argumnetem
;tej instrukcji jest adres powrotu. W programie jest to PC-0x0007. Czyli jeśli aktualna
;zawartość PC wynosi 0x005A to 0x005A - 0x0007 = 0x0053. To oznacza, że wykonanie
;programu zostaje przeniesione z powrotem na początek zasadniczej części kodu do
;instrukcji LDS
0000005A  RJMP PC-0x0007        ;Relative jump
```

Należy zwrócić uwagę na to, że zmienne przed dodaniem są ładowane do rejestrów i po tej operacji następuje sumowanie, po czym wynik jest umieszczany w pamięci/ zmiennej wynikowej, tu **c**. Pętla **while** jest zrealizowana, za pomocą instrukcji **RJCM** i adresem skoku wyrażonym, jako PC-0x0007, czyli 53 tj. początek pętli **while.**

Zadania do wykonania:

1. Omówiony w przykładzie przypadek proszę przeanalizować dla dwóch sytuacji: pierwsza – suma danych wejściowych jest mniejsza od 255 (a+b < 255) i druga – suma danych wejściowych jest większa od 255 (a+b > 255). Jak to wpływa na rejestr statusu ?
2. Eksperymenty z punktu pierwszego należy powtórzyć dla zmienionych typów zmiennych na **signed char**. Jaki jest domyślnie **char**, ze znakiem czy bez ?
3. Wracamy do definicji zmiennych jako **char**. Eksperymenty i analizę należy powtórzyć zmieniając definicję zmiennej WYJŚCIOWEJ (tylko tej !) z **char** na **int.** Zmiana wpływa na rozmiar z pojedynczego bajtu na dwa bajty. Ten przypadek proszę przeanalizować w dwóch wariantach: pierwszy – suma danych wejściowych (char) jest mniejsza od 255 (a+b < 255) i drugi – suma danych wejściowych (char) jest większa od 255 (a+b > 255) Jak to wpływa na rejestr statusu ?
4. Eksperymenty i analizę należy powtórzyć zmieniając definicje WSZYSTKICH zmiennych z **char** na **int.** Zmiana wpływa na rozmiar zmiennych z pojedynczego bajtu na dwa bajty.

Każdorazowo po zmianie typu danych należy ponownie wygenerować kod asemblerowy i przeanalizować kod programu.

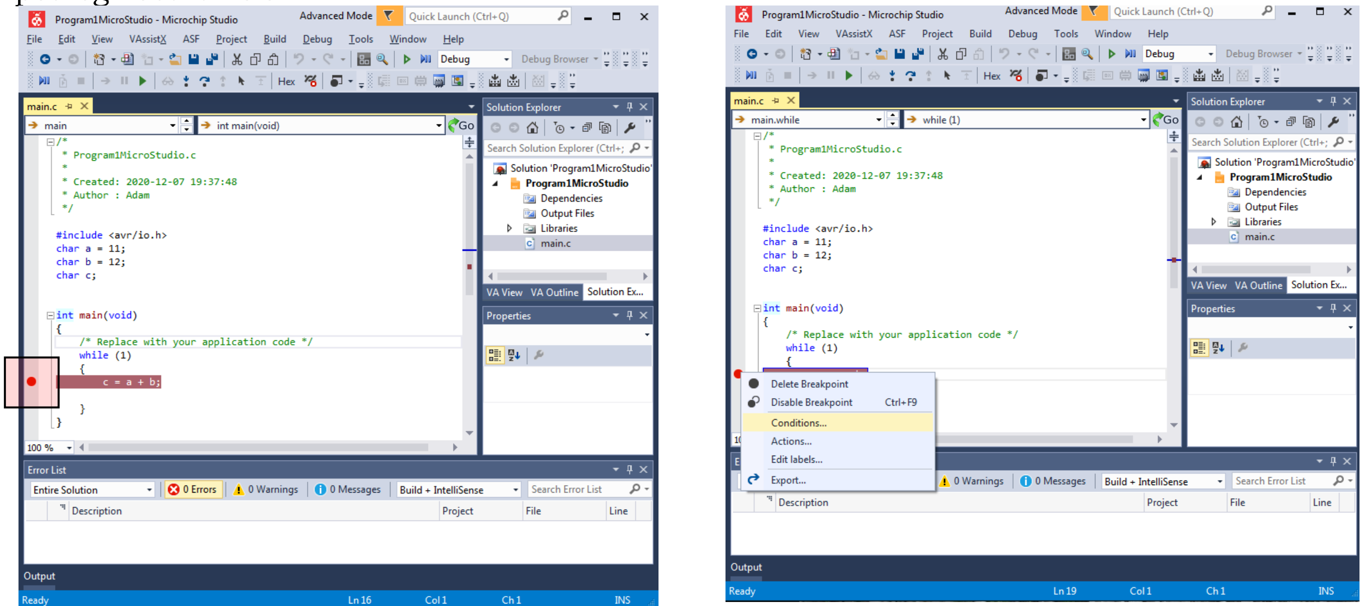
5. Eksperymenty czy analizę należy raz jeszcze powtórzyć zmieniając definicje zmiennych z **int** na **float**. W tym przypadku zmiany w programie asemblerowym są już znaczne. W tym przypadku należy przynajmniej określić rozmiar zmiennych. Dla zainteresowanych należy analiza dodawania zmiennych rzeczywistych.

6. Kolejna zmiana polegać będzie na dodaniu instrukcji inkrementacji:

```
c = a + b;
a++; //ewentualnie dla przyspieszenia: a += 10;
```

Przy założeniu typów zmiennych **char** i **int** należy każdorazowo wygenerować i przeanalizować kod asemblerowy.
Co się dzieje jeśli **c** będzie większe od 126? Obserwacja flag **SR** – **Status Register** będzie pomocna. Można zmienić definicję zmiennych na **unsigned char** i powtórzyć eksperymenty. Obserwacje warto przeprowadzić z zastosowaniem pułapek, ang. breakpoints. Taką pułapkę

można „zastawić” klikając z lewej strony w szarym obszarze okna z kodem (rysunek 4). Dodatkowo wywołując menu kontekstowe do pułapki można dodatkowo ustawić warunek (**Conditions…**). Przydatne jeśli np.: pętla ma wykonać 1000 iteracji, a sprawdzeniu/analizie podlega ostatnie 5.



Rysunek 4. Ustawianie pułapek.