Программирование на Си в среде CCS

Язык Си

- Язык высокого уровня
- Низкоуровневые механизмы обращения с данными
- Оптимально преобразуется в объектный исполняемый код
- Широкое распространение
- Очень большая поддержка

Программа на Си

```
/* объявление переменных */
int a; // описание первого слагаемого
int b; // описание второго слагаемого
int c; // описание результата (суммы)
/* основная программа */
void main() /* главная функция программы */
{
// текст программы – подсчёт суммы двух чисел
  a = 10;
  b = 5;
  c = a + b; // подсчёт суммы двух чисел
}
```

Программа на Си

Состоит из функции main(), которая является главной функцией проекта.

Содержит любое количество функций, которые вызываются из главной функции и из других функций.

В любом месте программы можно объявить переменные.

Главная функция

Функция «main» является главной функцией программы. Она обязательно должна присутствовать в проекте. После того, как будет выполнен код инициализации микроконтроллера и глобальных переменных программы, управление будет передано этой функции. Можно сказать, что выполнение пользовательской программы начинается с этой функции.

```
/* основная программа */
void main() /* главная функция программы */
{
/* текст программы – подсчёт суммы двух чисел */
  a = 10;
  b = 5;
  c = a + b; // подсчёт суммы двух чисел
}
```

Оператор присваивания

Оператор присваивания – «=». Записывает в переменную значение. Кроме того существует понятие «составного присваивания»: выполнить действие с переменной и сохранить результат в эту же переменную:

`x = x + 5` можно записать как `x+=5`

`y = y – 2` можно записать как `y-=2`

Объявление переменных

Сначала указывается тип переменной, затем её имя.

```
/* объявление переменных */
int a; // описание первого слагаемого
int b; // описание второго слагаемого
int c; // описание результата (суммы)
```

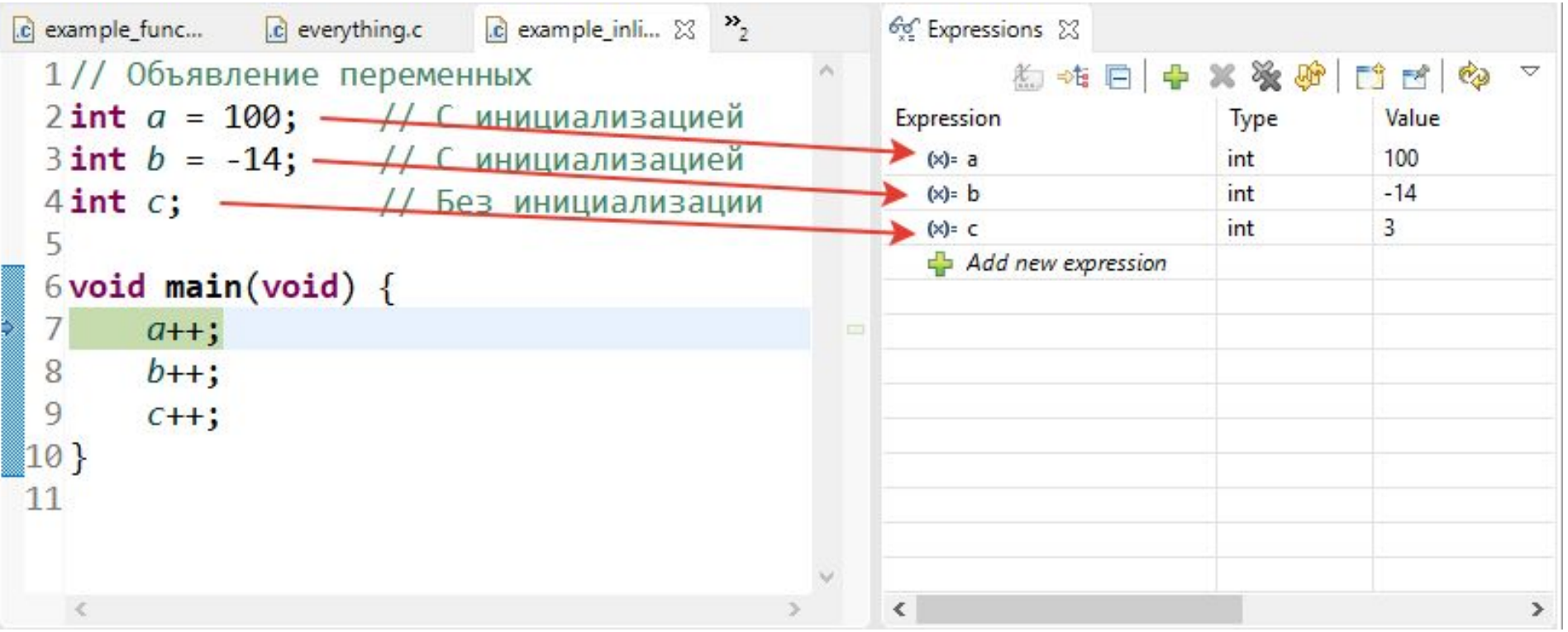
Можно объявить несколько переменных одного типа через запятую.

```
/* объявление переменных */
int a, b, c;
```

Объявление переменных

При объявлении переменной можно сразу её проинициализировать. В таком случае, к началу выполнения программы (функции «main») её значение будет известно.

Если переменную не проинициализировать, то к началу выполнения программы в ней может лежать что угодно – «мусор».



Имена переменных

- Состоят из букв и цифр
- Первый символ – всегда буква
- Символ «_» считается буквой
- Строчные буквы для имен переменных
- Заглавные для символических констант
- Нельзя использовать в качестве имен зарезервированные слова (if, else, for...)

Типы данных языка Си для TMS320x28xx

Тип	Размер	Минимум	Максимум
char	16 бит	-32768	32767
unsigned char	16 бит	0	65535
short	16 бит	-32768	32767
unsigned short	16 бит	0	65535
int	16 бит	-32768	32767
unsigned int	16 бит	0	65535
long	32 бита	-2147483648	2147483647
unsigned long	32 бита	0	4294967295
enum	16 бит	-32768	32767
float	32 бита	-1,19E-38	1,19E+38
double	32 бита	-1,19E-38	1,19E+38
long double	32 бита	-1,19E-38	1,19E+38

Объявление массивов

Массив – последовательность элементов одного типа в памяти. Синтаксис объявления массива: сначала указывается типа элементов массива, затем имя, а после имени в квадратных скобках – количество элементов.

Нумерация элементов начинается с «0»

```
// Объявление массива «a» из 100 элементов типа "int"
int a[100];

void fillBuf (void){
    a[0] = 5;   // Записать число "5" в первый элемент массива
    a[1] = 6;   // Записать число "6" во второй элемент массива
}
```

Структуры

Структуры позволяют сгруппировать несколько переменных разных типов. Также среди членов структуры могут быть указатели на переменные и указатели на функции. Пример описания структуры:

```
// Пример описания структуры с элементами разных типов
struct myStruct {
    int mem1;
    unsigned long mem2;
    long *pmem3;

    void (*someFunc)();
};
```

Структуры

Использование структур удобно, когда нужно задать несколько объектов, имеющий одинаковые свойства. Например – каналы АЦП, которые имеют коэффициент усиления, смещение сигнала, код АЦП, результат преобразования и т.д. Один контроллер может поддерживать 10 – 15 каналов АЦП, что может дать 45 переменных.

```
// Структура-описатель канала АЦП
struct adcChannel {
    // Коэфф. усиления
    int gain;
    // Смещение сигнала
    int offset;
    // Результат оцифровки
    int result;

    // Функция очистки данных
    void (*clear)();
};
```

```
void main (void) {
    // Создать структуры для обработки АЦП
    struct adcChannel Udc;          // Напр. ЗПТ
    struct adcChannel Ia, Ib, Ic;   // Фазные токи

    // Настройка канала АЦП
    Udc.gain = 540;
    Udc.offset = 0;
}
```

Инициализация структур

Структуру можно инициализировать двумя способами:

- Присвоить разные значения каждому элементу
- Присвоить всем элементам одно и то же значение

Значения для инициализации приводятся после знака « = » в фигурных скобках:

```
// Определение структуры даты
struct Sdate {
    int year, month, day, dayOfWeek;
    int hour, minute, second;
};

// Создание экземпляра. Каждый элемент инициализирован своим значением
struct Sdate dateOfBirth = {1980, 11, 21, 4, 13, 24, 11};
// Все элементы инициализированы значением "0"
struct Sdate dateOfDeath = {0};
```

Операции языка Си

- Арифметические
- Сравнения
- Логические
- Поразрядные логические
- Присваивания

Арифметические операции

Оператор	Операция
+	Сложение
-	Вычитание
*	Умножение
/	Деление
%	Остаток от деления
++	Инкремент
--	Декремент

Инкрементирование и декрементирование

Инкремент – увеличение значения переменной на 1

Декремент – уменьшение значения переменной на 1

Различают пред-инкремент и пост-инкремент:

```
x++;     // x увеличивается на 1 после использования
x--;     // x уменьшается на 1 после использования
++x;     // x увеличивается на 1 перед использованием
--x;     // x уменьшается на 1 перед использованием
```

Пост- и пред- инкремент

Постинкремент: сначала с переменной производятся все действия, описанные в выражении, после чего её значение увеличивается на 1.

```
void main(void) {
    a = 10;
    b = a++;
    // Сначала в b будет записано значение переменной a,
    // а затем "a" будет увеличена на 1.
    // В результате получится b = 10, a = 11
}
```

Прединкремент: перед началом всех действий переменная увеличивается на 1, а затем это увеличенное значение используется в выражении.

```
void main(void) {
    a = 10;
    b = ++a;
    // Сначала "a" будет увеличена на 1,
    // а затем в "b" будет записано значение переменной "a".
    // В результате получится b = 11, a = 11
}
```

Операция деления

При делении одного целого числа на другое результат получается ЦЕЛЫЙ, дробная часть отсекается:

```
void main(void) {
    int a = 5;
    int b = 2;
    int c;

    // Так как переменные "a", "b", "c" - целые (имеют тип int),
    // результат их деления также будет целым, без учёта остатка
    // Таким образом c = 2
    c = a / b;
}
```

Управление ходом вычислений

- `if else`
- `switch`
- `while`
- `do while`
- `for`
- `break`
- `continue`

if else

Иногда надо выбрать тот или иной вариант действий в зависимости от некоторых условий: если условие верно, поступать одним способом, а если неверно — другим. Для этого применяют условные операторы.
Один из них – «if … else» - имеет следующий вид:

```
if (выражение)
   инструкция1
else
   инструкция2
```

if else

Пример использования:

```
void checkCommands(void) {
    // Если пришла команда на выключение,
    // остановить привод, иначе пересчитать напряжение фаз
    if (commandStop == 1) {
        stopDrive();
    } else {
        updateVoltages();
    }
}
```

В общем случае часть «else {…}» может отсутствовать.

В теле оператора «if» или «else» может выполняться одно действие (как в примере выше), а может несколько. Если выполняется только одно действие, то фигурные скобки ставить не нужно, но всё равно рекомендуется.

else if

Также можно проверять несколько разных условий при помощи дополнительных блоков «else if (…)». Можно использовать сколько угодно таких проверок. В это случае будут выполнены только те действия, которые находятся внутри блока, условие которого истинно. Если истинны условия сразу в нескольких блоках, то будет выполнен только первый.

```
void main(void) {
    // Если пришла команда на включение, подготовить и
    // запустить привод; иначе, если пришла команда на
    // остановку, остановить привод. Иначе повторно
    // запросить команды
    if (commandStart == 1) {
        clearAllFaults();
        startDrive();
        updateVoltages();
    } else if (commandStop == 1) {
        stopDrive();
        setZeroVoltage();
    } else {
        requestCommands();
    }
}
```

Критерий истинности условия оператора «if»

Истинным считается любое значение, не равное нулю:

```
if (4)
   doCase1();
else if (100 / 8)
    doCase2();
```

Более привычным условием для оператора считается результат сравнения двух чисел/переменных:

```
// Если скорость двигателя меньше 100 об/мин,
// увеличить скорость; а если больше - уменьшить
if (driveSpeed < 100)
    driveSpeed += 5;
 else if (driveSpeed > 100)
    driveSpeed -= 5;
```

Операции сравнения

Оператор	Операция
>	Больше чем
>=	Больше или равно
<	Меньше чем
<=	Меньше или равно
==	Равно
!=	Не равно

Логические операции

Оператор	Операция
&&	И
||	Или
!	Не, отрицание

Поразрядные логические операции

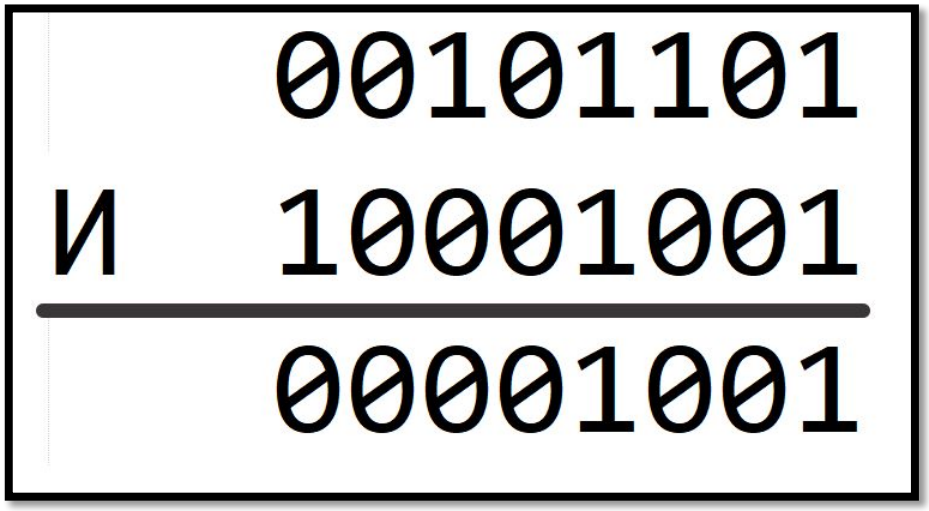
Оператор	Операция
&	побитовое И
|	побитовое Или
^	Исключающее Или
~	побитовое Не
>>	Сдвиг вправо
<<	Сдвиг влево

Поразрядные логические операции

Следует различать операции «&&» и «&».

Логическое И («&&») даёт результат «истина», когда оба операнда не равны 0, иначе даёт результат «ложь». Результат такой операции, как правило, используется только для условного ветвления программы («если (A && B), то выполнить действие_1»).

Побитовое И («&») даёт результат операции «И» между соответствующими битами переменных. Используется для различных целей, например для наложения маски:



Пример использования логических операторов

Операторы «логическое И» и «логическое ИЛИ» используются, когда какое-то действие нужно выполнить в том случае, когда выполняется сразу несколько условий (в случае оператора И) или, наоборот, хотя бы одно из условий (в случае оператора ИЛИ):

```
// Если нет аварий, задание скорости не нулевое
// и есть команда запуска - запустить привод
// А если есть авария или команда "стоп" - остановить
if (faultCounter == 0 && speedReference != 0 && commandStart == 1) {
    startDrive();
} else if (faultCounter != 0 || commandStop == 1) {
    stopDrive();
}
```

Приоритеты логических операций

Приоритет у «&&» выше чем у «||» и обе они младше операций отношения и равенства.

Выражение вычисляется в порядке приоритетов. Если результат выражения оказывается известен до окончания вычислений, то вычисления прекращаются. Это следует учитывать при программировании.

Операции отношения имеют приоритет перед операциями проверки равенства.

Арифметические операции старше операций отношения:

a < b – 1 эквивалентно a < (b – 1)

Цикл while

Цикл «while» имеет следующий формат вызова:

while (выражение){

    инструкции

}

Вычисляется выражение. Если его значение равно «истине», то выполняется инструкция. Этот цикл будет выполняться, пока выражение не станет ложным, после чего вычисления продолжатся с точки сразу за инструкцией.

```
// Вычитывание данных из массива до тех пор,
// пока он не опустеет
while (bufferEmptyFlag == 0) {
   readDataFromBuffer();
   bufferEmptyFlag = checkBuffer();
}
```

Цикл do while

Цикл «do while» имеет следующий формат вызова:

do

инструкция

while (выражение)

Выполняется инструкция. Вычисляется выражение. Если его значение равно «истине», то цикл будет выполняться. Отличается от цикла while() тем, что «инструкция» гарантированно будет выполнена хотя бы один раз, даже если «выражение» ложно.

```
// Пока не будет заполнен весь массив,
    // заполнять его случайными числами
    do {
        // Записать в элемент с номером "counter" случайное число
        buffer[counter] = randomNumber();
        // Увеличить счётчик элементов
        counter++;
    } while (counter < bufferSize);
```

Цикл for

Цикл «for» имеет следующий формат вызова:

```
for (выражение1; выражение2; выражение3){
        инструкция1;
}
```

Выражение1 выполняется только один раз, затем проверяется **выражение2**, и если оно истинно, выполняются **инструкции**. После выполнения всех инструкций выполняется **выражение3**. Затем проверяется **выражение2**, и если оно истинно выполняются инструкции и так далее.

Что эквивалентно конструкции:

```
выражение1;
while (выражение2) {
        инструкция1;
        выражение3; }
```

Цикл for

Цикл «for» обычно используется в тех случаях, когда заранее известно количество итераций. Наиболее часто для работы с массивами.
В данном примере объявляется вспомогательная переменная «i», которая используется для перебора элементов массива. В цикле «for» эта переменная вначале обнуляется (**выражение1: I = 0**), затем проверяется условие, что конец массива не достигнут (**выражение2: i < bufferSize),** затем обнуляется элемент массива с номером **i**. После этого переменная **i** инкрементируется (**выражение3: i++**). Таким образом все элементы массива от 0 до 99 будут обнулены.

```
// Очистить массив элементов
    int i;
    for (i = 0; i < bufferSize; i++) {
        buffer[i] = 0;
    }
```

switch и break

Оператор «**switch**» позволяет выполнять те или иные действия, в зависимости от значения переменной. Синтаксис оператора:

```
switch (переменная){
        case значение1:
                инструкция1;
        break;

        case значение2:
                инструкция2;
        break;

        case значение3:
                инструкция3;
        break;

}
```

Оператор switch и break

```
// В зависимости от значения переменной driveMode,
// запустить тот или иной алгоритм управления
switch (driveMode){
    case 0:
        scalarMode();
    break;

    case 1:
        vectorMode();
    break;

    case 2:
        slowDown();
    break;

    default:
        doNoting();
    break;
}
```

Наша первая программа на Си

```
/* Наша первая программа на языке СИ */
/* Определение и инициализация глобальных переменных
*/
int x = 2;
int y = 7;
/* Основная программа */
void main (void){
    // Определение локальной переменной
    long z;
    // Секция выполнения – бесконечный цикл
    for( ; ; ){
        z = x;
        z = z + y; //Вычисление переменной z
    }
}
```

Функции

Служат для описания законченного алгоритма. С помощью функций задача разбивается на подзадачи, с целью упрощения отладки и возможности распараллелить процесс написания проекта между программистами.

Формат функции

тип_результата *имя_функции* (**список_аргументов**) {
    тело функции
}

```
// Функция возведения переменной в степень
// Аргументы: x - число, которое нужно возвести
//            pow - требуемая степень
long power (int x, unsigned int pow) {
    unsigned int i;      // Вспомогательная переменная
    long result = x;     // Результат

    // Умножаем результат на x столько раз,
    // сколько требуется
    for (i = 1; i < pow; i++){
        result = result * x;
    }

    // Возвращаем результат
    return result;
}
```

Вызов функции

Чтобы вызвать функцию, необходимо написать её имя и передать соответствующие аргументы. В примере выше функция называлась «power» и принимала два аргумента. Типы переменных которые передаются в функцию должны соответствовать типам аргументов. Это также касается и результата.

```
void main(void) {
    int varA = 20;
    unsigned int powerA = 8;
    long res;

    // Записать в res число, которое вернёт вернёт функция power
    // В результате получится res = varA в степени powerA
    res = power(varA, powerA);
}
```

Аргументы функции

Переменные, которые передаются в функцию в качестве аргументов **не меняются** внутри функции, как бы она не была устроена внутри. Когда в функцию передаются переменные, то внутри неё временно **создаются их копии**, и вся работа внутри функции **происходит с копиями**.

```
// Функция умножения числа на 100
int multiple100 (int x) {
    // Чтобы не занимать стек локальными переменными,
    // не будем создавать переменную "result", а изменим
    // само значение аргумента:
    x = x * 100;
    return x;
}

void main(void) {
    int varA = 20;
    int varB;

    // Умножим переменную varA на 100 и запишем результат в varB
    varB = multiple100(varA);
    // В результате этой операции varB = 20000,
    // А значение varA не изменится: varA = 20
}
```

Указатели

Си поддерживает низкоуровневое неконтролируемое обращение к памяти с помощью указателей.

int *px;

Данная строка кода создаст переменную типа указатель.

То есть px хранит в себе адрес, а *px – содержимое, размещенное по этому адресу.

```
// Объявить переменную varA и указатель, который пока что
// не указывает ни на какую переменную
int varA = 1;
int *pointerVarA;

void main(void) {
    // Записать в указатель "pointerVarA" адрес переменно varA
    // Символ "&" означает "получить адрес"
    pointerVarA = &varA;

}
```

Операции с указателями

Основные операции над указателями: увеличение (инкремент), уменьшение (декремент) и присваивание. Увеличить или уменьшить указатель можно не только операциями инкремента/декремента, но и сложением с константой.

Важно понимать, что **инкрементирование/декрементирование** указателя меняет его значение так, чтобы он указывал на **следующий/предыдущий** элемент. Таким образом, если указатель ***px*** указывает на переменную ***x*** типа **int**, которая занимает 1 ячейку памяти, то операция «***px***++» увеличит значение указателя на 1. А если бы переменная ***x*** имела тип **long**, который требует 2 ячейки памяти, то та же операция увеличила бы указатель ***px*** на 2.

Поэтому тип указателя должен совпадать с типом переменной, на которую он указывает. То есть указатель, который объявлен как «int *p», не может указывать на переменную, которая объявлена как «long x».

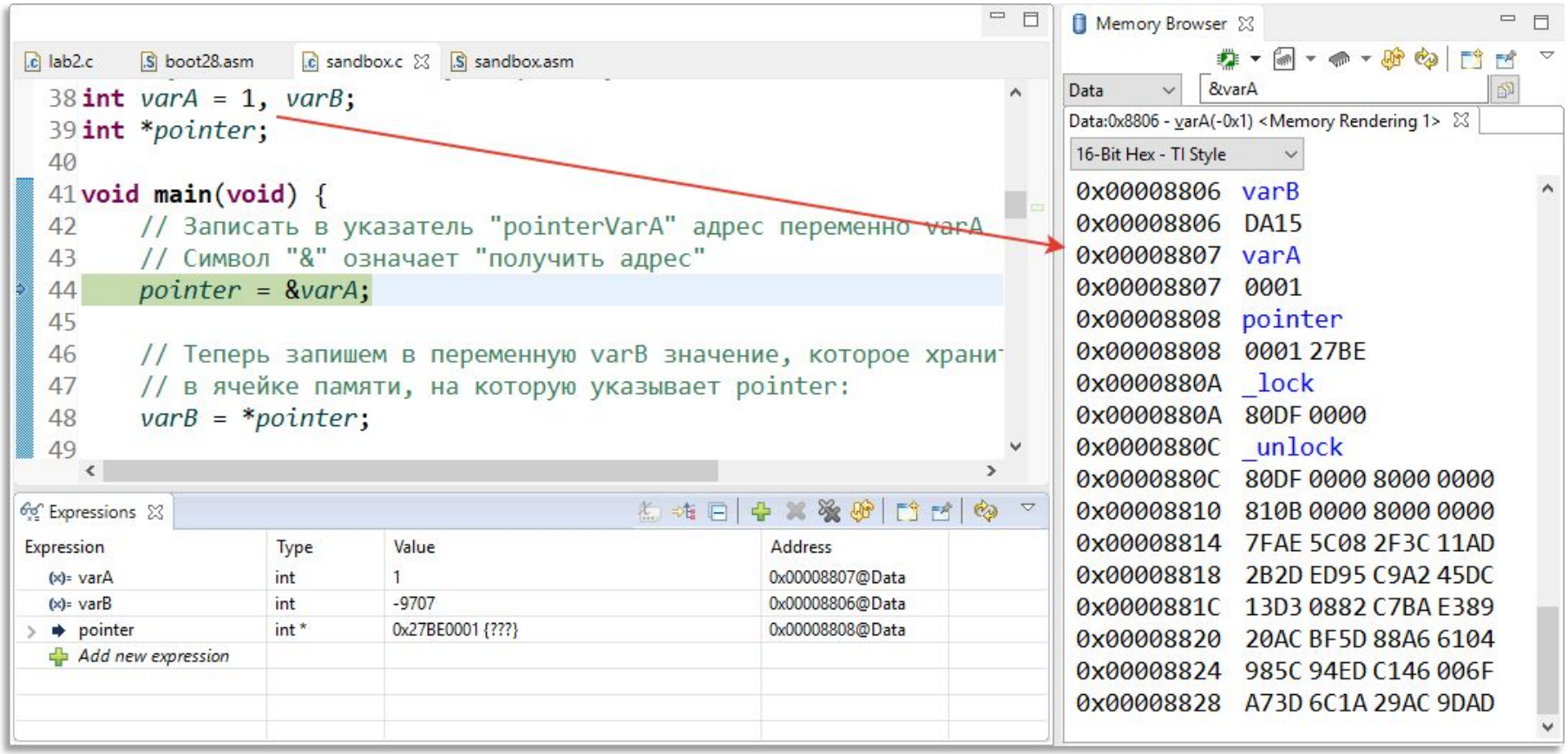
Указатели

Также через указатель можно получить значение, которое хранится в той ячейке, на которую он указывает, при помощи оператора « * »:

```
// Объявить переменные varA, varB и указатель, который пока что
// не указывает ни на какую переменную
int varA = 1, varB;
int *pointer;

void main(void) {
    // Записать в указатель "pointerVarA" адрес переменно varA
    // Символ "&" означает "получить адрес"
    pointer = &varA;

    // Теперь запишем в переменную varB значение, которое хранится
    // в ячейке памяти, на которую указывает pointer:
    varB = *pointer;

    // Так как pointer указывал на переменную varA, то теперь
    // varB имеет такое же значение: varB = 1
}
```

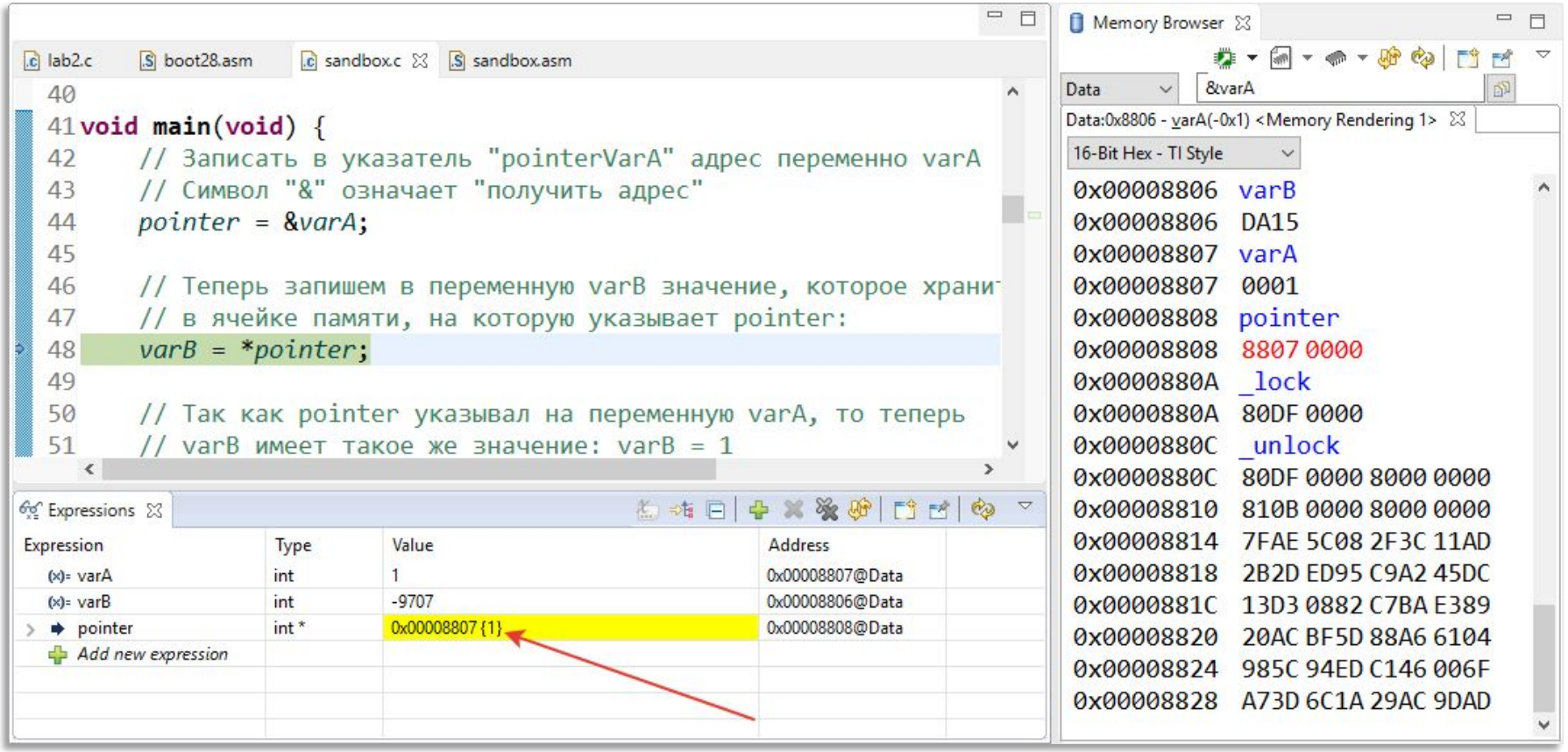
Указатели - пример

Выполнение программы из предыдущего слайда.
Шаг первый: переменная «**varA**» находится по адресу 0x8807,
Указатель «**pointer**» пока что указывает на какую-то случайную ячейку



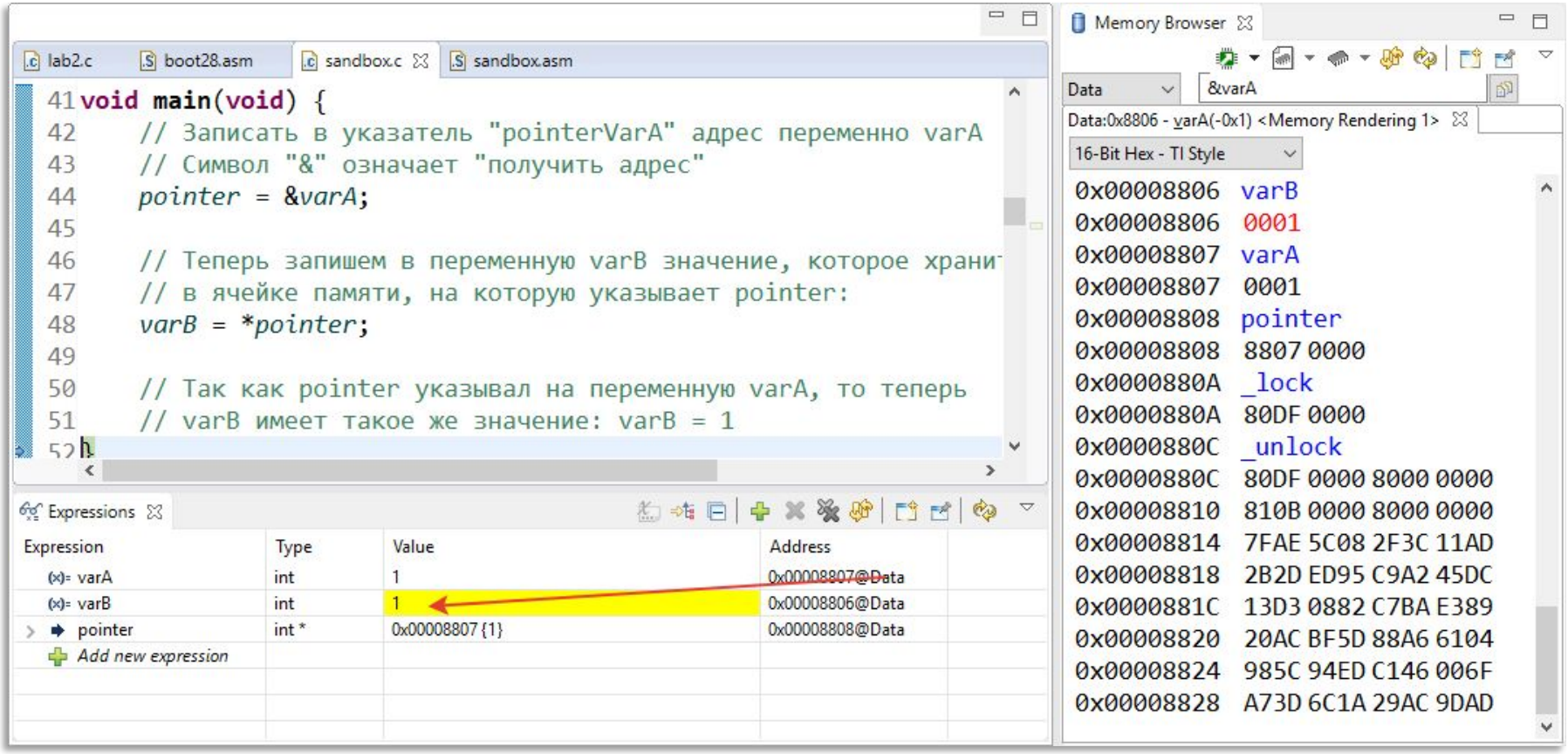
Указатели - пример

Шаг второй: в указатель «**pointer**» записан адрес переменной «**varA**»



Указатели - пример

Шаг третий: в переменную **«varB»** записано значение ячейки памяти, на которую указывал **«pointer».** Т.к. он указывал на переменную **«varA»,** то значение «**varB**» теперь равно значению «**varA**».



Указатели и массивы

Указатели часто используются для перебора массивов

```
int inputBuf[100];
int *input;
int x, y;

void main(){
    // Имя массива является указателем на его первый элемент
    // Следующие две строки идентичны
    input = inputBuf;
    input = &inputBuf[0];

    // В переменную x запишется первый элемент массива,
    // а в y - второй
    x = *input;
    y = *(input + 1);

    x = inputBuf[0];
    y = inputBuf[1];
}
```

Указатели и структуры

Также может быть создан указатель на *структуру*. В таком случае указатель хранит адрес первого элемента структуры. При обращении к элементам структуры через указатель, используется оператор « -> » вместо « . »

```
struct my_time systime; // Структура
struct my_time *t;      // Указатель на структуру
void main (void) {
    // Доступ к элементу структуры через имя экземпляра
    systime.hours = 1;

    // Доступ к элементу структуры через указатель
    t->hours = 1;
}
```

Указатели и функции

Иногда удобно передавать данные в функцию через указатели. Это удобно, когда необходимо возвращать значения сразу нескольких переменных. Например, если функция должна записать значение в переменную и вернуть какой-то результат своей работы:

```
// Функция чтения значения из регистра
int readValue (long* x){
    // Если буфер пуст, вернуть -1
    // Иначе записать значение и вернуть 0
    if (bufferEmpty == 1) {
        return -1;
    } else {
        *x = ethernetData;
        return 0;
    }
}
```

Указатели и функции

Если аргументом функции является указатель, то передавать нужно адрес переменной, а не саму переменную.
В этом случае, значение переменной будет изменено (в отличие от случая, когда передаётся сама переменная).

```
void main (void) {
    int result, data;

    // Вызываем функция, передавая в неё адрес переменной
    // Если функция была выполнена успешно - обрабатываем данные
    result = readValue(&data);
    if (result == 0) {
        processData();
    }
}
```

Определение функций в отдельном файле

При создании проекта разными программистами или фирмами удобно создавать различные функции в разных исходных файлах. Такой подход позволяет получать готовые для компоновки объектные файлы, которые могут распространяться без перекомпиляции и раскрытия исходного кода.

По умолчанию все функции Си являются глобальными. Если происходит вызов глобальной функции, то ее описание через прототип необязательно. Описание требуют только внешние переменные используемого модуля. Для этого создаются заголовочные файлы, содержащие информацию об объектном файле библиотеки.

Квалификатор extern

Используется при модульном программировании

```
extern volatile int    x;
……
volatile int           x=7;
```

Этот квалификатор означает, что в одном из файлов **определена** переменная « x ».

Если при объявлении выделяется память под переменную, то процесс называется определением. *Использование extern приводит к объявлению, но не к определению*. Оно просто говорит компилятору, что определение происходит где-то в другом месте программы.

При описании переменной с этим квалификатором, *под неё не выделяется места в памяти*, но компилятор знает тип этой переменной и как правильно совершать с ней различные действия.

Заголовочные файлы .h

Файл.h должен содержать полную информацию по используемому модулю.

Хотя это и необязательно, но принято описывать прототипы функций модуля, чтобы программист, пользующийся модулем знал названия включенных в него функций и формат вызова каждой из них.

Будучи включенным в программу, заголовочный файл должен описывать все, что может потребоваться программисту для обращения к функциям модуля и переменным как ко внутренним функциям проекта.

Подключаемые файлы

Директива #include включает указанный файл в текущую позицию компилируемого файла. Имя файла заключается либо в «” ”», либо в «< >».

Имя файла может содержать полный или частичный путь к файлу или не содержать пути.

#include <stdio.h>

#include “main.h”

#include “C:\...\main.h”

Фактически эта директива заменяется на содержимое включаемого файла.

Пример создания внешнего модуля

Модуль должен производить суммирование двух целых чисел. Входными и выходными переменными являются переменные модуля.

```
// Файл “functions.c”
// Функция умножения
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long mult (int a, int b) {
    return a * b;
}

// Функция деления
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long div (int a, int b) {
    return a / b;
}
```

Пример создания внешнего модуля

Заголовочный файл:

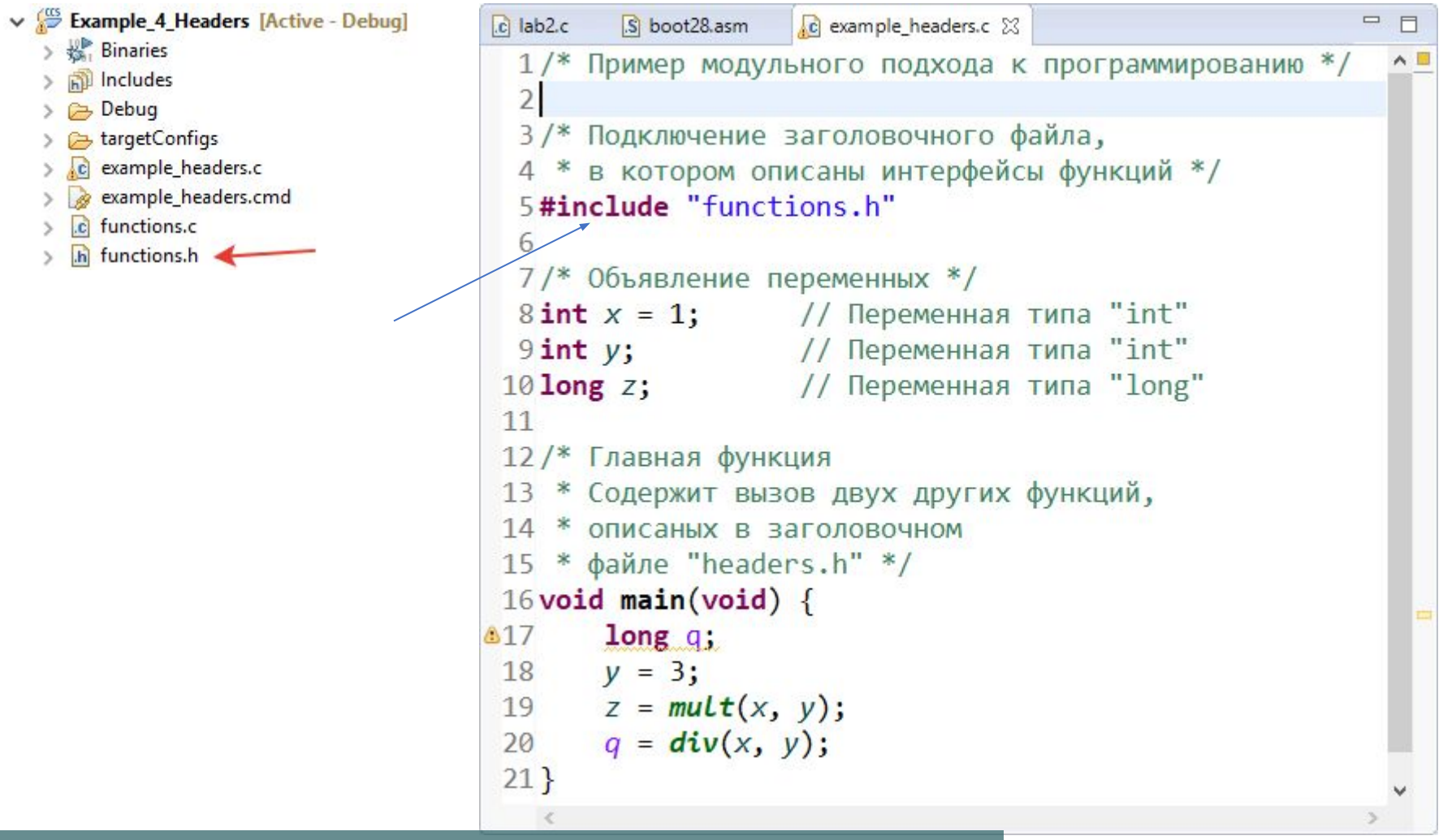
```
/* Заголовочный файл, в котором
 * описан интерфейс функций
 */

#ifndef FUNCTIONS_H_
#define FUNCTIONS_H_

long mult (int, int);
long div (int, int);

#endif /* FUNCTIONS_H_ */
```

Пример создания внешнего модуля

Основная программа, использующая модуль:

```
/* Пример модульного подхода к программированию */
/* Подключение заголовочного файла, в котором описаны интерфейсы функций
*/
#include "functions.h"

/* Объявление переменных */
int x = 1;        // Переменная типа "int"
int y;            // Переменная типа "int"
long z;           // Переменная типа "long"

/* Главная функция. Содержит вызов двух других функций,
 * описанных в заголовочном файле "headers.h" */
void main(void) {
    long q;
    y = 3;
    z = mult(x, y);
    q = div(x, y);
}
```

Пример создания внешнего модуля



Пример создания внешнего модуля

lab2.c | example_hea... | functions.h

```c
/* Заголовочный файл, в котором
 * описан интерфейс функций
 */

#ifndef FUNCTIONS_H_
#define FUNCTIONS_H_

long mult (int, int);
long div (int, int);

#endif /* FUNCTIONS_H_ */

```

functions.c

```c
 * functions.c
 *
 *  Created on: 16 авг. 2017 г.
 *      Author: Dmitry
 */

// Функция умножения
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long mult (int a, int b) {
    return a * b;
}

// Функция деления
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long div (int a, int b) {
    return a / b;
}

```

Пример создания внешнего модуля

lab2.c | boot28.asm | example_headers.c

```c
/* Пример модульного подхода к программированию */

/* Подключение заголовочного файла,
 * в котором описаны интерфейсы функций */
#include "functions.h"

/* Объявление переменных */
int x = 1;       // Переменная типа "int"
int y;           // Переменная типа "int"
long z;          // Переменная типа "long"

/* Главная функция
 * Содержит вызов двух других функций,
 * описаных в заголовочном
 * файле "headers.h" */
void main(void) {
    long q;
    y = 3;
    z = mult(x, y);
    q = div(x, y);
}

```

functions.c

```c
/*
 * functions.c
 *
 *  Created on: 16 авг. 2017 г.
 *      Author: Dmitry
 */

// Функция умножения
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long mult (int a, int b) {
    return a * b;
}

// Функция деления
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long div (int a, int b) {
    return a / b;
}

```

Пример создания внешнего модуля

lab2.c | boot28.asm | example_headers.c

```c
/* Пример модульного подхода к программированию */

/* Подключение заголовочного файла,
 * в котором описаны интерфейсы функций */
#include "functions.h"

/* Объявление переменных */
int x = 1;       // Переменная типа "int"
int y;           // Переменная типа "int"
long z;          // Переменная типа "long"

/* Главная функция
 * Содержит вызов двух других функций,
 * описаных в заголовочном
 * файле "headers.h" */
void main(void) {
    long q;
    y = 3;
    z = mult(x, y);
    q = div(x, y);
}

```

functions.c

```c
/*
 * functions.c
 *
 *  Created on: 16 авг. 2017 г.
 *      Author: Dmitry
 */

// Функция умножения
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long mult (int a, int b) {
    return a * b;
}

// Функция деления
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long div (int a, int b) {
    return a / b;
}

```

Пример создания внешнего модуля

lab2.c | boot28.asm | example_headers.c

```c
/* Пример модульного подхода к программированию */

/* Подключение заголовочного файла,
 * в котором описаны интерфейсы функций */
#include "functions.h"

/* Объявление переменных */
int x = 1;       // Переменная типа "int"
int y;           // Переменная типа "int"
long z;          // Переменная типа "long"

/* Главная функция
 * Содержит вызов двух других функций,
 * описаных в заголовочном
 * файле "headers.h" */
void main(void) {
    long q;
    y = 3;
    z = mult(x, y);
    q = div(x, y);
}

```

functions.c

```c
/*
 * functions.c
 *
 *  Created on: 16 авг. 2017 г.
 *      Author: Dmitry
 */

// Функция умножения
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long mult (int a, int b) {
    return a * b;
}

// Функция деления
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long div (int a, int b) {
    return a / b;
}

```

Пример создания внешнего модуля

lab2.c | boot28.asm | example_headers.c

```c
/* Пример модульного подхода к программированию */

/* Подключение заголовочного файла,
 * в котором описаны интерфейсы функций */
#include "functions.h"

/* Объявление переменных */
int x = 1;       // Переменная типа "int"
int y;           // Переменная типа "int"
long z;          // Переменная типа "long"

/* Главная функция
 * Содержит вызов двух других функций,
 * описаных в заголовочном
 * файле "headers.h" */
void main(void) {
    long q;
    y = 3;
    z = mult(x, y);
    q = div(x, y);
}

```

functions.c

```c
/*
 * functions.c
 *
 *  Created on: 16 авг. 2017 г.
 *      Author: Dmitry
 */

// Функция умножения
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long mult (int a, int b) {
    return a * b;
}

// Функция деления
// В качестве аргументов принимает два числа типа int
// возвращает число типа long
long div (int a, int b) {
    return a / b;
}

```

Пример заголовочного файла

```
/* ===========================================================================
File name:       CLARKE.H  (IQ version)
Originator: Digital Control Systems Group
            Texas Instruments
Description:
Header file containing constants, data type definitions, and function prototypes for the
CLARKE
History:
05-15-2002      Release Rev 1.0                                          */
typedef struct {    _iq  as;        /* Input: phase-a stator variable  */
    _iq  bs;            /* Input: phase-b stator variable  */
    _iq  ds;            /* Output: stationary d-axis stator variable  */
    _iq  qs;            /* Output: stationary q-axis stator variable  */
    void  (*calc)();    /* Pointer to calculation function */
    } CLARKE;
typedef CLARKE *CLARKE_handle;
/*------------ Default initalizer for the CLARKE object. ---*/
#define CLARKE_DEFAULTS { 0, 0, 0, 0, (void (*)(long))clarke_calc }
/*----------- Prototypes for the functions in CLARKE.C ---*/
void clarke_calc(CLARKE_handle);
```

Включаемые функции

Когда вызывается включаемая функция, то ее код вставляется непосредственно в то место программы, где она вызывается.

Преимущества включаемых функций:

- Экономия времени на операциях вызова и возврата из функции
- Включенная функция может оптимизироваться компилятором совместно с окружающим ее кодом

Типы включаемых функций:

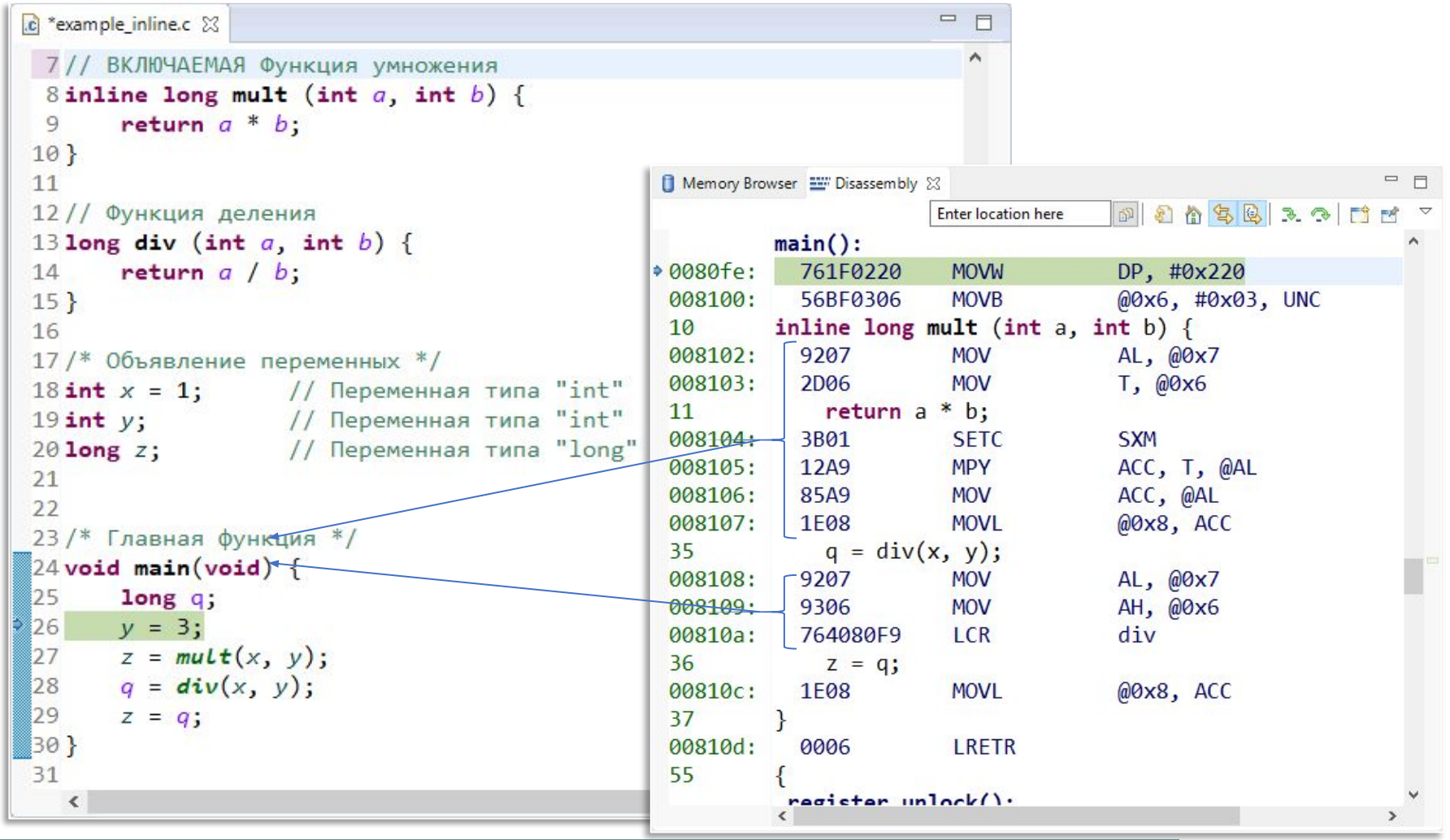
Встроенные операторы (встроенные операторы всегда включаемые (+, -, *, /...))

Автоматическое включение

Управляемое включение при определении функции

Включаемые функции

Включение функций осуществляется ключевым словом `inline`



Директива #define

Директива #define является одной из директив препроцессора. Она позволяет создать макроопределение, которое можно использовать в коде программы.
Перед компиляцией исходного файла выполняется препроцессинг, который обрабатывает директивы (#define, #include, #pragma и другие). Во время препроцессинга все встречающиеся макроопределения в коде будут заменены на соответствующие значения.

```
// Определить число Пи
#define PI  3.1416

void main (void) {
    // Задать радиус и рассчитать площадь круга
    float radius = 4;
    float square = radius * radius * PI;
}
```

Директива #define

Важно понимать, что макроопределение это не переменная. С помощью этой директивы можно определять не только константы, а любые конструкции. Потому что фактически препроцессор просто заменит имена макроопределений на их значения. В примере ниже слово «condition» будет заменено на набор условий:

```
// Макроопределение сложного условия
#define condition (forceRun == 1) || \
                  ((startCommand > 0) && (enableRun == 1) \
                          && (faults == 0))

void main (void) {
    // Проверить условие запуска двигателя:
    // либо есть команда принудительного запуска (forceRun),
    // Либо есть команда обычного запуска, при этом нет аварий
    // и работа разрешена
    if (condition)
        startDrive();
}
```

Директива #define

Исходный файл:

```
#define condition (startCommand == 1)

void main (void) {
    if (condition)
        startDrive();
}
```

После препроцессинга:

```
void main (void) {
    if ((startCommand == 1))
        startDrive();
}
```

Методы условной компиляции. #ifdef и #ifndef

Метод условной компиляции состоит в использовании директив ***#ifdef*** и ***#ifndef***, что соответственно означает «если определено» и «если не определено». Стандартный вид ***#ifdef*** следующий:

#ifdef имя_макроса
последовательность операторов
#endif

Если имя макроса определено ранее в операторе ***#define***, то последовательность операторов, стоящих между ***#ifdef*** и ***#endif***, будет компилироваться.

Стандартный вид ***#ifndef*** следующий:

#ifndef имя_макроса
последовательность операторов
#endif

Если имя макроса не определено ранее в операторе ***#define***, то последовательность операторов, стоящих между ***#ifndef*** и ***#endif***, будет компилироваться.

Методы условной компиляции.

```
280
281 //#define PWM_TEST
282
283 //! Нормирование входных величин.
284 //!Учитывает компенсацию напрЯженияЯ при изменении Ud,
285 //!выполнЯет вписывание вектора в окружность, если требуетсЯ, и
286 //!выполнЯет смену базиса, относительно которого идет нормировка.
287 //! \memberof TVectPWM
288 void PWM_NORM_INPUT(TPWM_6_12* p) {
289
290     p->UalfaNorm = p->UalfaRef;
291     p->UbetaNorm = p->UbetaRef;
292
293
294 #ifdef PWM_TEST
295     //находим амплитуду (без учета ограниченияЯ)
296     p->U_mag = _IQmag(p->UalfaNorm, p->UbetaNorm);
297
298     p->UdCorTmp = _IQdiv(_IQ(1.0),
299             (_IQ(1.0)+_IQmpy((adc.Udc_meas-_IQ(1.0)),p->UdCompK)));
300     //учет пульсаций напрЯженияЯ на Ud
301     if (p->UdCompEnable & 1) { //он включен?
302         p->UalfaNorm = _IQmpy(p->UalfaNorm, p->UdCorTmp); //изменим пропорционально коэфф-ту коррекции
303         p->UbetaNorm = _IQmpy(p->UbetaNorm, p->UdCorTmp); //и это тоже
304     }
305 #endif
306     //вписывание заданной амплитуды напрЯженияЯ в окружность, вписываемую в шестиугольник базывых векторов
307     //когда U_lim=1.0, это и есть такаЯ окружность. Бывает, что мы хотим вписывать в шестиугольник. Тогда про
308     if (p->U_lim > _IQ(1.0 / 0.866)) //но нет смысла задирать выше максимально-реализ. напрЯженияЯ (больше баз
309         p->U_lim = _IQ(1.0 / 0.866);
```

Методы условной компиляции.

```
281 #define PWM_TEST
282
283 //! Нормирование входных величин.
284 //!Учитывает компенсацию напрЯженияЯ при изменении Ud,
285 //!выполнЯет вписывание вектора в окружность, если требуетсЯ, и
286 //!выполнЯет смену базиса, относительно которого идет нормировка.
287 //! \memberof TVectPWM
288 void PWM_NORM_INPUT(TPWM_6_12* p) {
289
290     p->UalfaNorm = p->UalfaRef;
291     p->UbetaNorm = p->UbetaRef;
292
293
294 #ifdef PWM_TEST
295     //находим амплитуду (без учета ограниченияЯ)
296     p->U_mag = _IQmag(p->UalfaNorm, p->UbetaNorm);
297
298     p->UdCorTmp = _IQdiv(_IQ(1.0),
299             (_IQ(1.0)+_IQmpy((adc.Udc_meas-_IQ(1.0)),p->UdCompK)));
300     //учет пульсаций напрЯженияЯ на Ud
301     if (p->UdCompEnable & 1) { //он включен?
302         p->UalfaNorm = _IQmpy(p->UalfaNorm, p->UdCorTmp); //изменим пропорционально коэфф-ту коррекции
303         p->UbetaNorm = _IQmpy(p->UbetaNorm, p->UdCorTmp); //и это тоже
304     }
305 #endif
306     //вписывание заданной амплитуды напрЯженияЯ в окружность, вписываемую в шестиугольник базывых векторов
```

Использование оптимизатора

- Оптимизация «–o0». Оптимизация уровня регистров.
  - Располагает переменные непосредственно в регистры
  - Осуществляет оптимизацию циклов программы
  - Удаляет неиспользованный программой код
  - Упрощает арифметические выражения и выходную отчетность
  - Вставляет в программу напрямую текст функций объявленных «inline»

Используется если ваш модуль имеет функции, которые вызываются из других модулей и глобальные переменные, которые изменяются в других модулях.

Использование оптимизатора

- Оптимизация «–o1». Выполняет те же действия, что при оптимизации «-o0», и дополнительно:
  - осуществляет прямое присвоение локальных констант;
  - удаляет неиспользованные присвоения переменных;
  - исключает локальные повторяющиеся выражения;

Оптимизация используется если Ваш модуль не имеет функций, которые вызываются другими модулями, но имеет глобальные переменные, которые изменяются в других модулях.

Использование оптимизатора

- Оптимизация «–o2». Выполняет те же действия, что и при оптимизации «-o1», и дополнительно:
  - осуществляет улучшенную оптимизацию циклов;
  - удаляет глобальные повторяющиеся подвыражения;
  - исключает глобальные повторяющиеся присвоения;

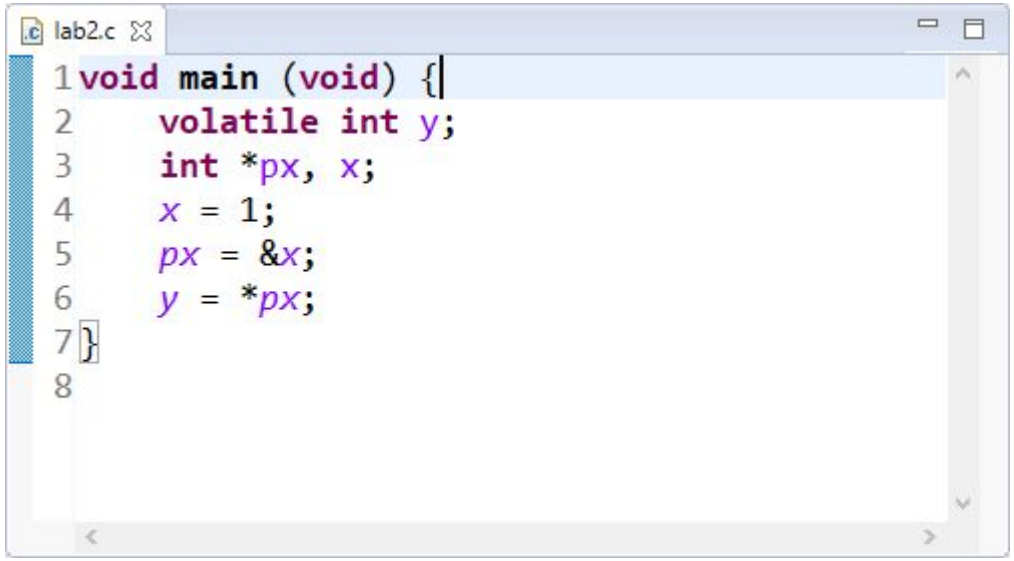
Оптимизатор использует «–o2» по умолчанию, даже если вы не указали уровень оптимизации. Оптимизация используется если ваш модуль не имеет функций, которые вызываются другими модулями или глобальных переменных, которые изменяются в других модулях

Использование оптимизатора

- Оптимизация «–o3». Выполняет те же действия, что и при оптимизации «-o2», и дополнительно:
  - удаляет все функции, которые не вызываются;
  - упрощает функции результат вычислений (присваивается return) которой не используется программой;
  - напрямую без вызова вставляет небольшие функции указанные директивой «inline»
  - перегруппирует описание функции, так чтобы атрибуты, вызываемой функции,

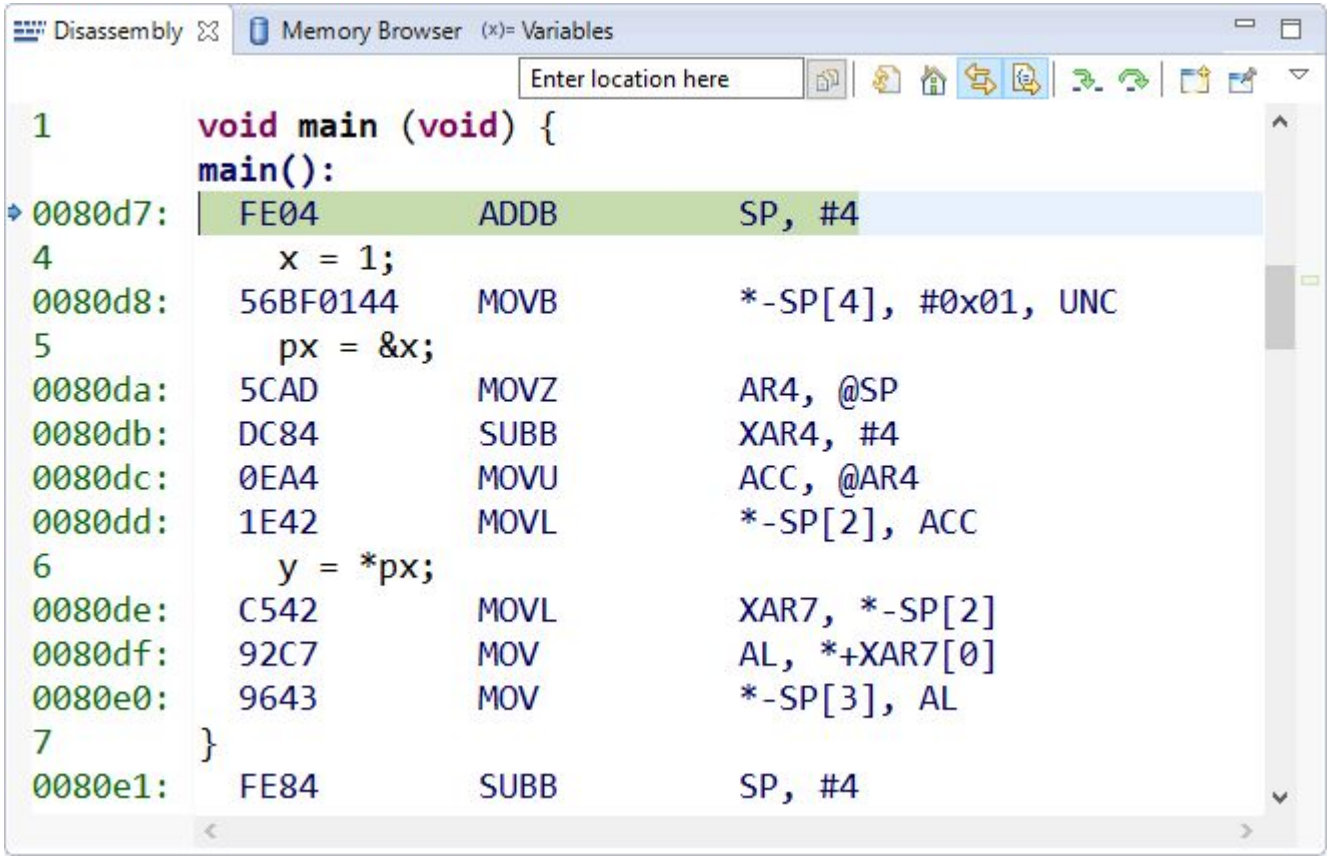
Использование оптимизатора

Исходная программа:



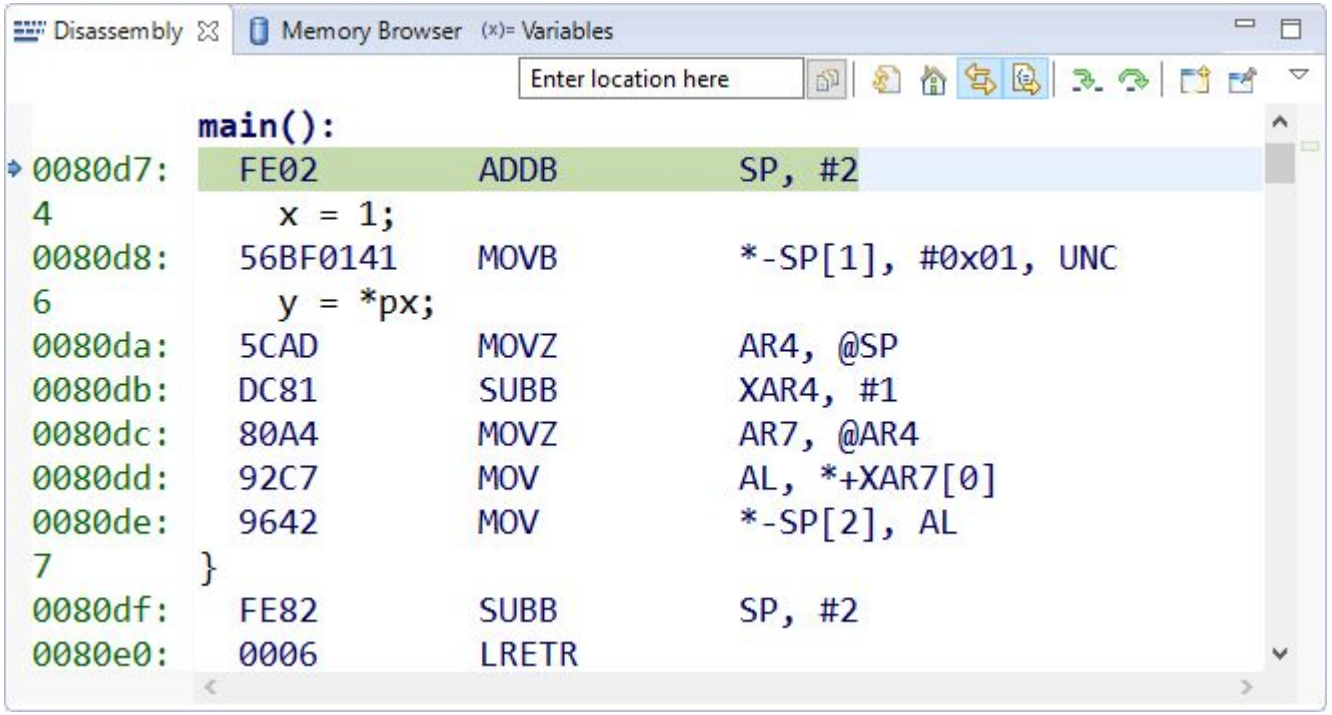
Использование оптимизатора

Без оптимизации



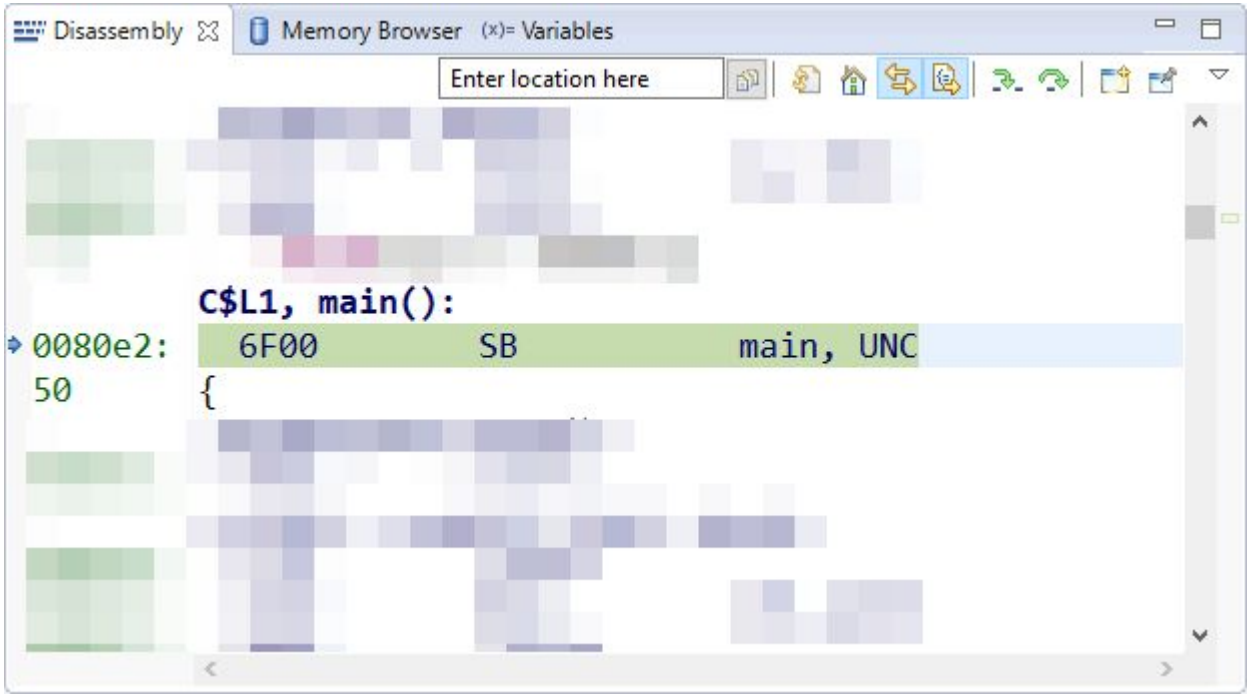
Использование оптимизатора

Первый уровень оптимизации (-o0)



Использование оптимизатора

Второй уровень оптимизации (-o1)



Компилятору очевидно, что `&x != 0x00FF` всегда

Опасность оптимизации

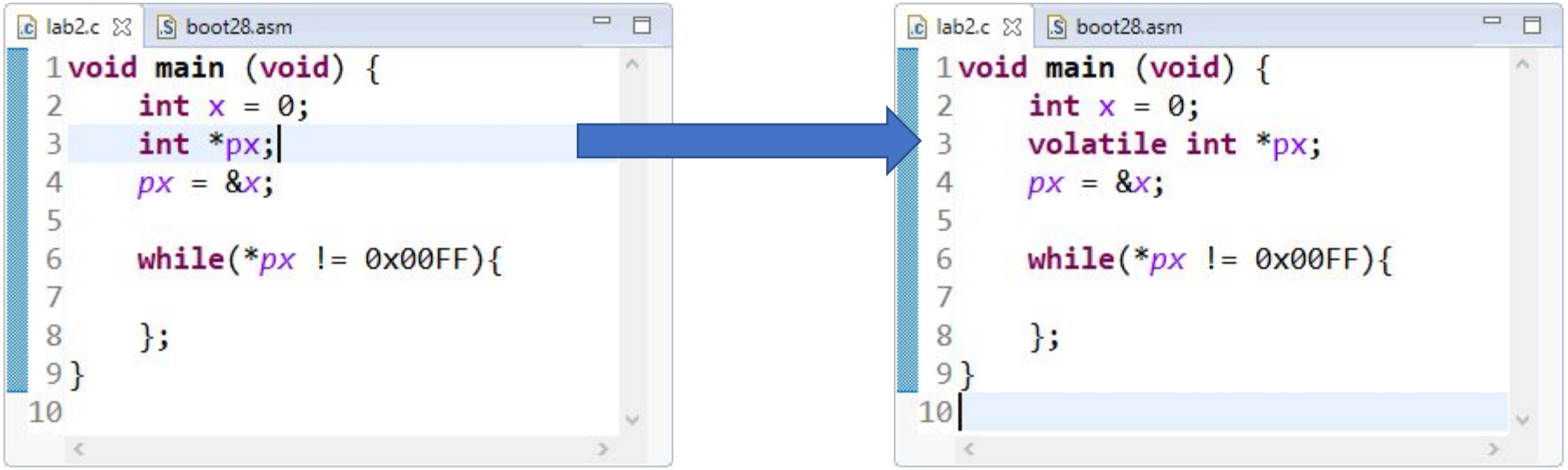
Многие периферийные регистры изменяются не программным путём. Их состояние зависит от состояния дискретных входов и выходов микроконтроллера. Компилятор видит, что переменная, отвечающая за периферию, не изменяется ни в одном месте программы, и может исключить её обработку.

Пример, когда возможна такая ситуация (состояние дискретного входа GPIO10 зависит от уровня сигнала на соответствующей ножке микроконтроллера):

```
41 // Проверка защиты привода
42 void checkProtection (void) {
43     // Проверить наличие сигнала аварии на входе микроконтроллера
44     // Если сигнал есть - вызвать функцию остановки привода
45     if (GpioG1DataRegs.GPADAT.bit.GPIO10 == 1) {
46         stopDrive();
47     }
48 }
```

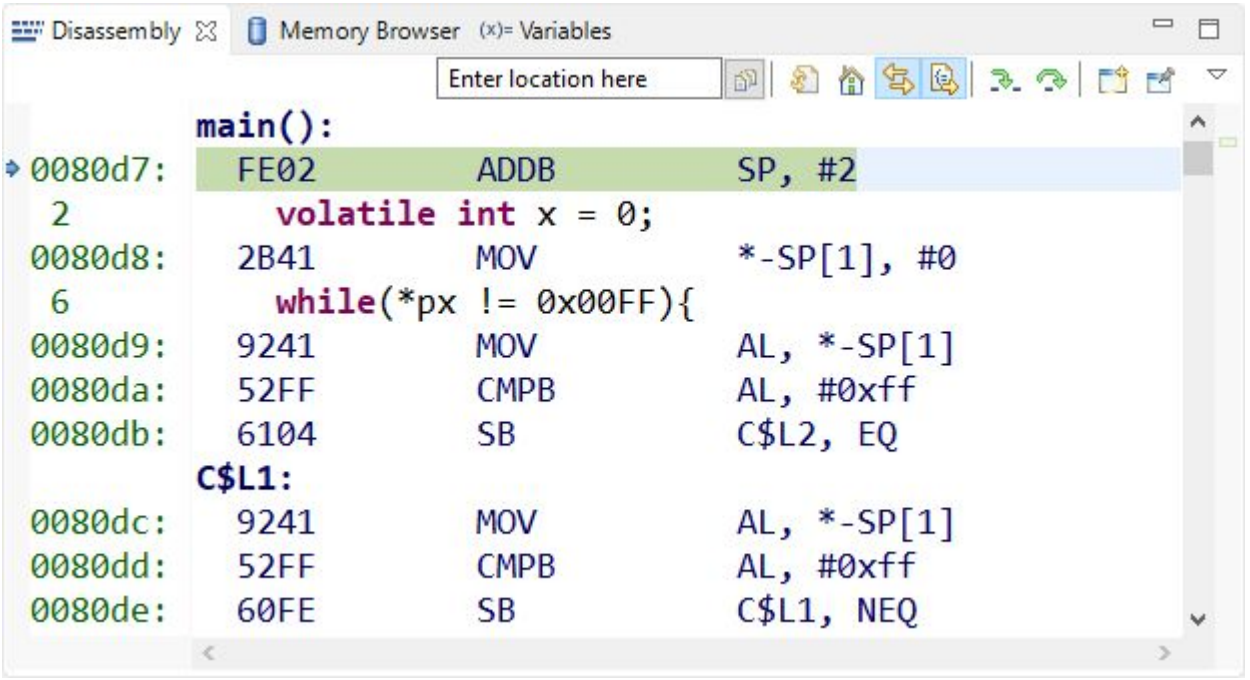
Использование оптимизатора

Использование ключевого слова “volatile” защищает чтение и запись переменной от оптимизации



Использование оптимизатора

Второй уровень оптимизации с использованием “volatile”



Оптимизация переменных

Использование статических переменных.

lab2.c | boot28.asm

```
1 void main (void) {
2     int a;
3     int b;
4     int c;
5
6     a = 1;
7     b = 2;
8     c = a + b;
9 }
```

Disassembly | Memory Browser | (x)= Variables

Enter location here

```
main():
0080d7:  FE04        ADDB        SP, #4
 6           a = 1;
0080d8:  56BF0141    MOVB        *-SP[1], #0x01, UNC
 7           b = 2;
0080da:  56BF0242    MOVB        *-SP[2], #0x02, UNC
 8           c = a + b;
0080dc:  9242        MOV         AL, *-SP[2]
0080dd:  9441        ADD         AL, *-SP[1]
0080de:  9643        MOV         *-SP[3], AL
 9       }
```

Оптимизация переменных

Использование статических переменных.

lab2.c | boot28.asm

```
1 void main (void) {
2     static int a;
3     static int b;
4     static int c;
5
6     a = 1;
7     b = 2;
8     c = a + b;
9 }
```

Disassembly | Memory Browser | (x)= Variables

Enter location here

```
 6          a = 1;
           main():
0080d7:    761F0220    MOVW    DP, #0x220
0080d9:    56BF0107    MOVB    @0x7, #0x01, UNC
 7          b = 2;
0080db:    56BF0206    MOVB    @0x6, #0x02, UNC
 8          c = a + b;
0080dd:    9206        MOV     AL, @0x6
0080de:    9407        ADD     AL, @0x7
0080df:    9608        MOV     @0x8, AL
 9      }
```

Оптимизация переменных

Статические переменные:

- не создаются при входе и не удаляются при выходе из функции (экономия процессорного времени)
- не расходуют пространство в стеке
- требуют больше памяти

Спасибо за внимание