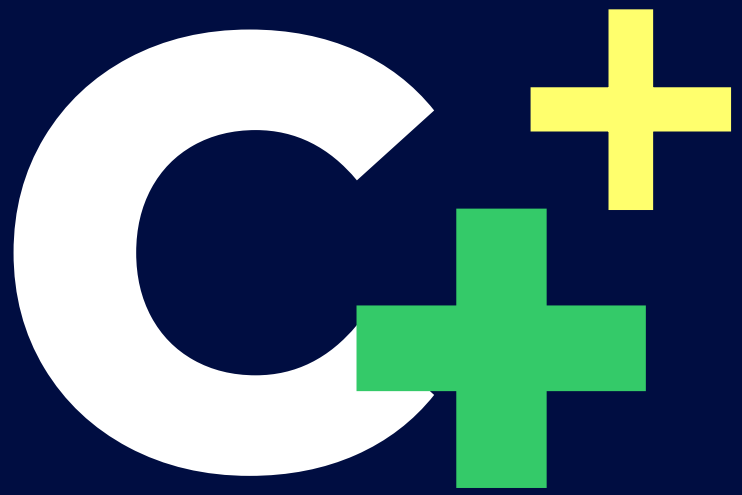


Создание среды разработки для C++

Взгляд изнутри

Профессиональная конференция для embedded - разработчиков C++

МИР

МИЛАНДР ГРУППА КОМПАНИЙ

МИР

Мастерская
инструментов
разработки

Подводные камни LLVM

или как создать компилятор для архитектуры с 32 -битным char'ом

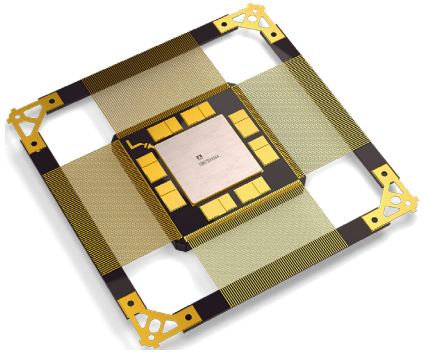
Алексей Спирков

Июнь 2019

В начале было

MILANDR

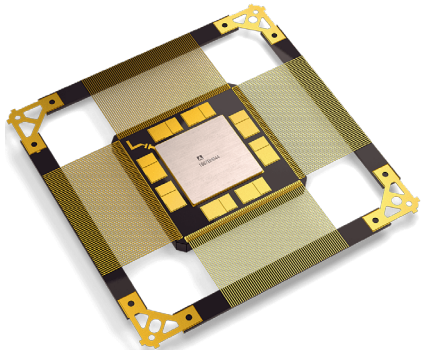
1967ВН028



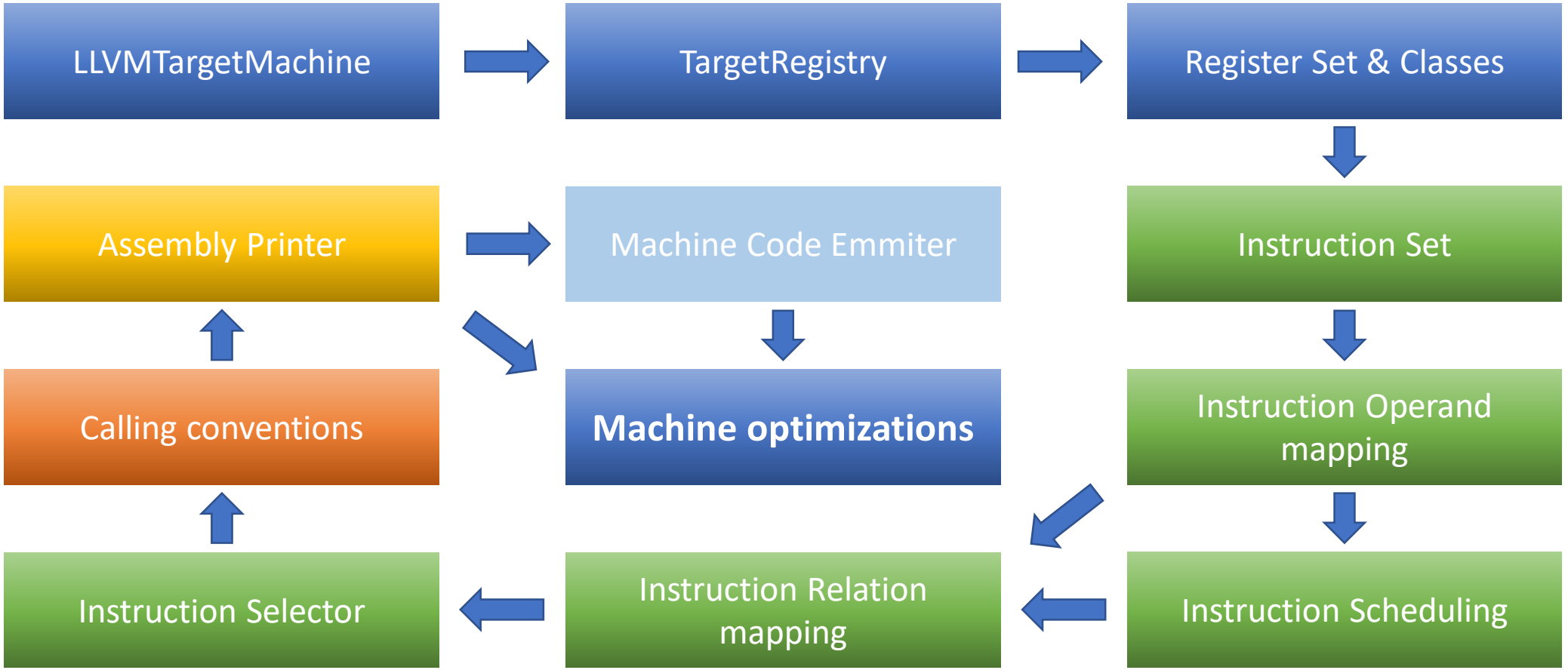
1967ВН044

- DSP функциональность
- Два вычислительных модуля
- Два блока целочисленных ALU
- Устройство управления
- Четыре 128-битные шины
- SOC-интерфейс
- Периферия
- 12Мбит внутренней памяти
- VLIW (до 4х инструкций за такт)

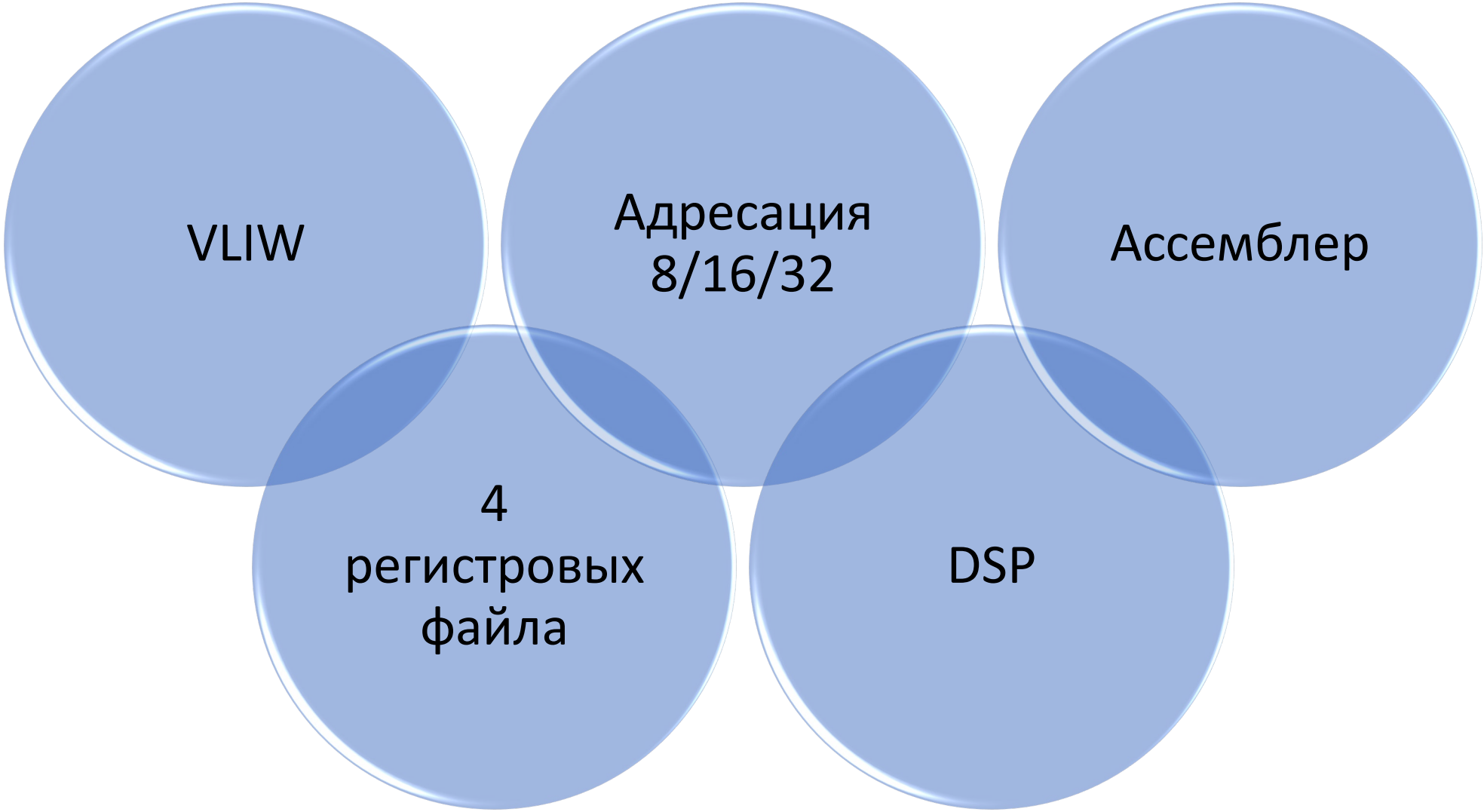
Компилятор



Базовый алгоритм портирования компилятора LLVM clang



Особенности целевой архитектуры



Точка зрения стандарта

Objects declared as characters (`char`) shall be large enough to store any member of the implementation's basic character set. If a character from this set is stored in a character object, the integral value of that character object is equal to the value of the single character literal form of that character. It is implementation-defined whether a `char` object can hold negative values. Characters can be explicitly declared `unsigned` or `signed`. Plain `char`, `signed char`, and `unsigned char` are three distinct types. A `char`, a `signed char`, and an `unsigned char` occupy the same amount of storage and have the same alignment requirements (3.9); that is, they have the same object representation. For character types, all bits of the object representation participate in the value representation. For unsigned character types, all possible bit patterns of the value representation represent numbers. These requirements do not hold for other types. In any particular implementation, a plain `char` object can take on either the same values as a `signed char` or an `unsigned char`; which one is implementation-defined.

There are four *signed integer types*: "`signed char`", "`short int`", "`int`", and "`long int`." In this list, each type provides at least as much storage as those preceding it in the list. Plain `int`s have the natural size suggested by the architecture of the execution environment[39] ; the other signed integer types are provided to meet special needs.

For each of the signed integer types, there exists a corresponding (but different) *unsigned integer type*: "`unsigned char`", "`unsigned short int`", "`unsigned int`", and "`unsigned long int`," each of which occupies the same amount of storage and has the same alignment requirements (3.9) as the corresponding signed integer type[40] ; that is, each signed integer type has the same object representation as its corresponding unsigned integer type. The range of nonnegative values of a *signed integer* type is a subrange of the corresponding *unsigned integer* type, and the value representation of each corresponding signed/unsigned type shall be the same.

Unsigned integers, declared `unsigned`, shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.[41]

Казалось бы – все просто

Но не тут то было!

Типичные ошибки LLVM

```
  if (!SrcVal->getType()->isIntegerTy())
    SrcVal = Builder.CreateBitCast(SrcVal, IntegerType::get(Ctx, StoreSize*8));

  // Shift the bits to the least significant depending on endianness.
  unsigned ShiftAmt;
  if (DL.isLittleEndian())
    ShiftAmt = Offset*8;
  else
    ShiftAmt = (StoreSize-LoadSize-Offset)*8;
```

```
    if (ElementSizeInBits % 8 != 0) {
      // GEPs over non-multiple of 8 size vector elements are invalid.
      return nullptr;
    }
    APInt ElementSize(Offset.getBitWidth(), ElementSizeInBits / 8);
    APInt NumSkippedElements = Offset.sdiv(ElementSize);
```

```
  Type *SplitIntTy =
      Type::getIntNTy(Ty->getContext(), NumElements * ElementSize * 8);
```

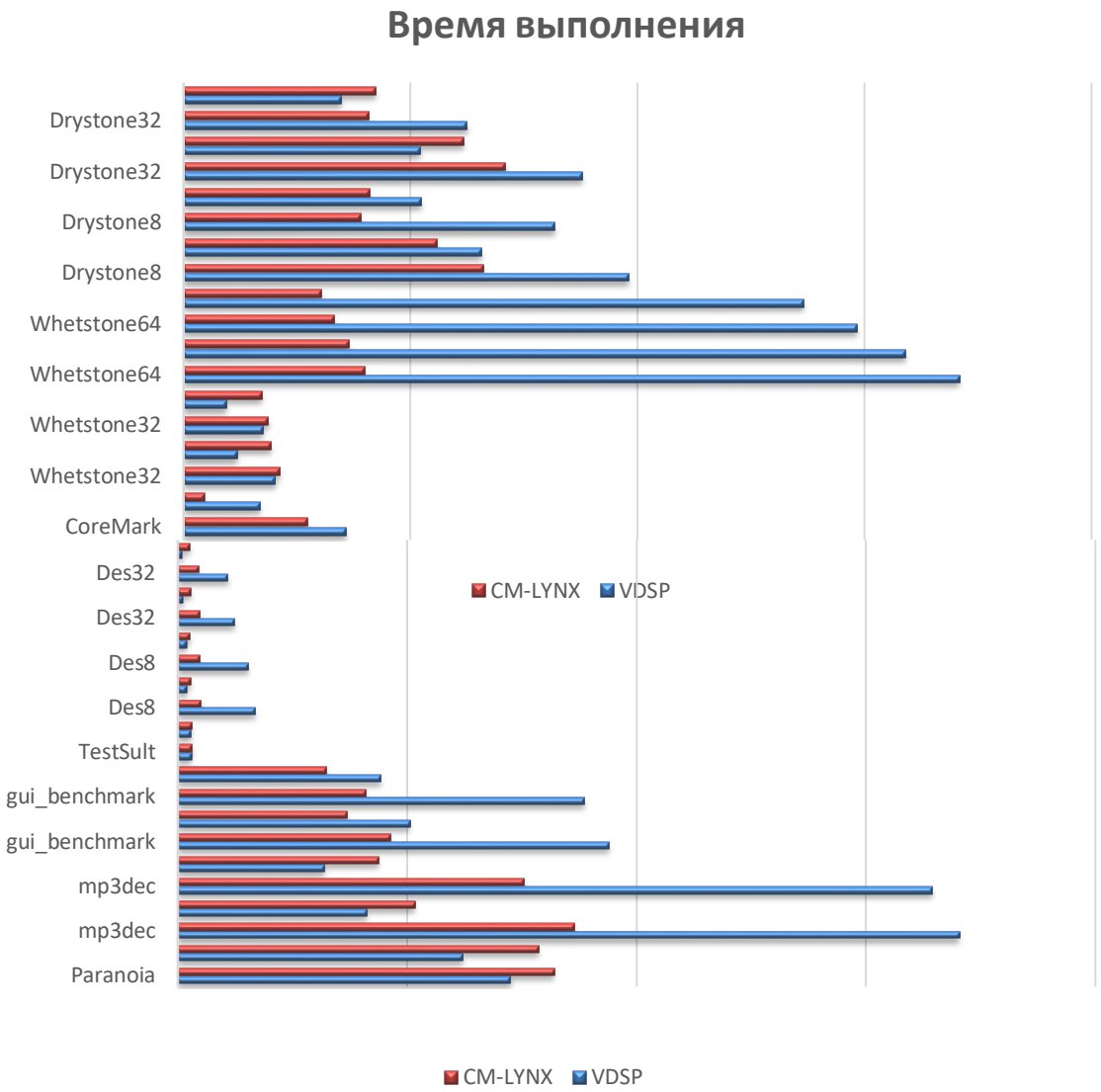
```
    TargetInfo::ConstraintInfo Info(Literal->getString(), OutputName);

    if (!Context.getTargetInfo().validateOutputConstraint(Info))
      return StmtError(Diag(Literal->getLocStart(),
```

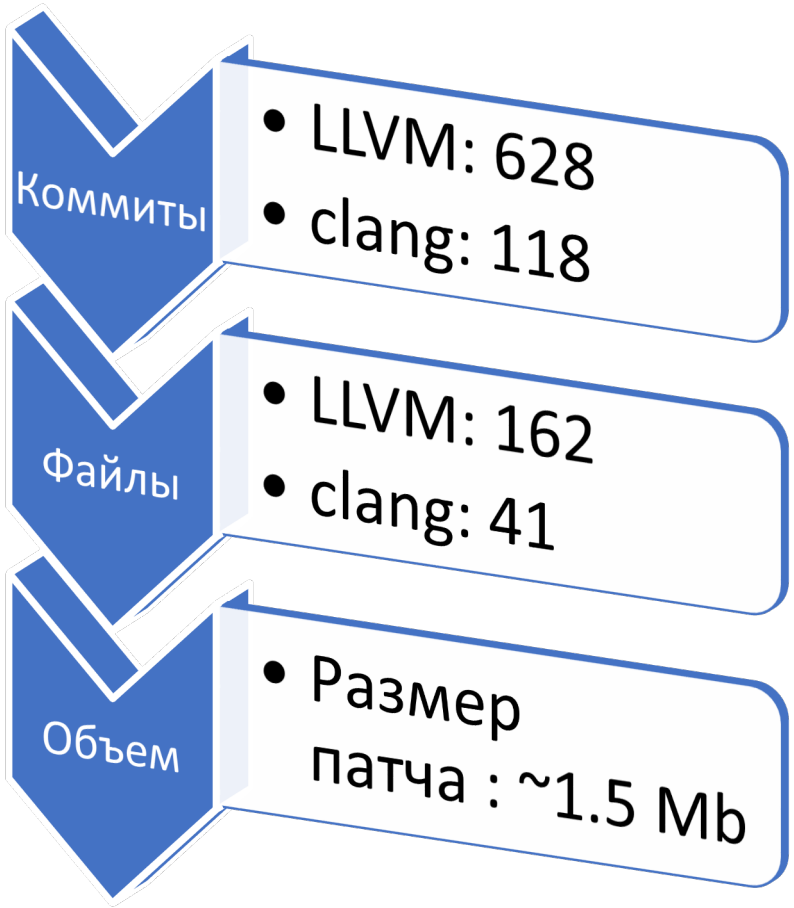
```
MVT TargetLoweringBase::getScalarShiftAmountTy(const DataLayout &DL,
                                               EVT) const {
  return MVT::getIntegerVT(8 * DL.getPointerSize(0));
}
```

А еще нужна поддержка 8 -ми битного char

- Адрес для 32 битной загрузки/сохранения – в 32х битных словах
- Для 16 битной – в 16ти битных
- Для 8 битной – в 8ми битной
- Спец режимы для кросс доступа

А в сухом остатке …





Вопросы?

Мастерская
инструментов
разработки

Алексей Спирков

hi@mir.astrosoft.ru