МАТЕРИАЛ 3. **СИСТЕМА МИКРОПРОГРАММИРОВАНИЯ СЕМЕЙСТВА MCS-51 ОТ KEIL ELEKTRONIK GMBH.**

Цель – изучить основные приёмы работы с программными кросс-средствами Keil Elektronik GmbH для разработки ПО микропроцессоров семейства MCS-51 и производных.

ВВЕДЕНИЕ.

Система микропрограммирования (СМ) - есть набор компактных программных средств для разработки, создания, отладки и сопровождения программ пользователя для микропроцессоров. СМ, как правило, построена для работы на ряде более мощных компьютеров, чем однокристальная МикроЭВМ. СМ обязательно включает программы (кросс-) **ассемблера** и **компилятора**, **редактора связей** (линкера), соответствующих **библиотек** языка высокого уровня, программного **симулятора** (отладчика) и библиотекаря. Дополнительно – эмулятор с его поддержкой, система реального времени и средства визуализации.

Отличия. 1). Для семейства x86 IA-16 и IA-32 – машина фон-Неймана (суперконвеерность, суперскалярность, спекулятивность). MCS-51 – раздельное управление памятью. 2). Для x86 – инверсный порядок байт (мл.б., ст.б.). Стек уменьшает адреса. MCS-51 - "старший байт - по младшему адресу". Стек увеличивает адреса. 3). Здесь стек ограничен. 4). Для x86 – выравнивание на границу существенно (2, 4, 16). MCS-51 - упаковка до байта. (XA – 2.)

Инструментальные средства системы программирования Keil Electronic GmbH 6.1:

UV2.exe – интегрированная среда µVision2 с комбинированным менеджером проекта и многофункциональным редактором текста;
S8051.DLL – многофункциональный отладчик (программный симулятор), версия 2.10;
DP51.DLL – ведение диалогов, версия 2.10;
Monitor-51 – программа отладки аппаратных средств с помощью µVision2 по последовательному каналу;
C51.exe – компилятор Си, версия 6.10;
A51.exe – макроассемблер, версия 6.10;
BL51.exe – редактор связей, версия 4.03;
LIB51.exe – библиотекарь, версия 4.13;
OH51.exe – 16-ый конвертер, версия 2.5;
RTX51TNY.lib – RTX51 Tiny – ядро ОС реального времени без поддержки приоритетов;
RTX51 Full – полное ядро ОС реального времени.

Система программирования MCS-51:
<-
Система программирования расширений MCS-51:
AX51 – макроассемблер;
CX51 – компилятор ANSI C;
LX51 – редактор связей;
LIBX51 – библиотекарь объектных модулей;
OHX51 – 16-ый конвертор.
Система программирования Intel/Temic 251:
A251 – макроассемблер;
C251 – компилятор ANSI C;
L251 – редактор связей;
LIB251 – библиотекарь объектных модулей;
OH251 – 16-ый конвертор.
Система программирования C166.
. . . . .

ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ Си.

Компилятор полностью поддерживает стандарт языка Си, выработанный Американским Национальным Институтом Стандартов (ANSI). Отличия от стандарта предназначены для лучшей поддержки семейства MCS-51. Компилятор написан полностью на VC++, не использует временных файлов и оверлеев и генерирует непосредственно стандартные объектные файлы. **Характеристики компилятора Си:** - соответствие стандарту ANSI; - несколько возможностей оптимизации: по скорости, по размеру или без оптимизации; - генерация кода, пригодного для зашивки в ПЗУ; - простой интерфейс с ассемблерными модулями; - тщательная проверка типов данных, производящаяся компилятором; - проверка интерфейса модулей, производящаяся линкером; - поддерживает системы с переключением банков (прозрачно для Си); - легко читаемые диагностические сообщения; - широкий набор опций управления листингом и выходными файлами; - библиотеки стандартных функций; - арифметика с плавающей точкой, совместимая со стандартом IEEE; - три модели памяти языка Си.

Компилятор специализирован под особенности семейства MCS-51 с её программной моделью.

Программная модель MCS-51.

Организация памяти. **Пространство памяти.**

Память программ имеет 16-битовую адресную шину, её элементы адресуются с использованием счётчика команд (PC) или инструкций, которые вырабатывают 16-разрядные адреса. Ряд МК содержат расположенную на кристалле внутреннюю память программ ёмкостью 4 Кбайт, которая может быть расширена за счёт внешней памяти программ. С точки зрения программиста имеется только один вид памяти программ объёмом 64 К. Для работы с памятью программ существуют специальные команды чтения **MOVC**.

Внутренняя память данных состоит из двух областей: 128 байт оперативной памяти с адресами 0–7Fh и области регистров специальных функций, занимающих адреса 80–FFh. Младшие 32 байта сгруппированы в 4 банка по 8 регистров общего назначения. Рабочий банк выбирается битами RS0 и RS1 в регистре PSW.

Память программ (внутр. + внешняя) (внешняя только)	Внешняя память данных (до 64К).	Внутренняя память данных. (пр. и косв. адр.)	**SFR** (только прямая адресация)	(8032AH, 8052AH, 8752BH, 80C51FA, …) (косв. адр.)
FFFF сегмент **CODE** до 64 К 0000	FFFF сегмент **XDATA** до 64 К 0000	007F сегмент **DATA** 128 байт 0000	00FF регистры специальных функций 0080	00FF сегмент **IDATA** 256 байт 0000

Внешняя память данных формируется дополнительными микросхемами памяти, подключаемыми к МК и может иметь ёмкость до 64 Кбайт. Для работы с внешней памятью данных существуют специальные команды чтения и записи **MOVX** (через DPTR или @R0, @R1).

Различия в размере кода и в скорости выполнения программы, определяются тем, что для доступа к данным из различных областей памяти используется разная последовательность кода.

```
int i;
i = 536;   /* 0x218
```

Таблица 1.

	DATA	IDATA	XDATA

Код	MOV i,#2
	MOV i+1,#18
		MOV R0,#i
		MOV @R0,#2
		INC R0
		MOV @R0,#18
			MOV DPTR,#i
			MOV A,#2
			MOVX @DPTR,A
			INC DPTR
			MOV A,#18
			MOVX @DPTR,A

Байты/циклы			
Диапазон	0-7F	0-FF	0-FFFF

Битовая адресация ОЗУ и регистров специальных функций. (**BIT**). Битовый процессор.

	(**DATA**)							
	. . . . .							
2FH	7F	7E	7D	7C	7B	7A	79	78
2EH	77	76	75	74	73	72	71	70
2DH	6F	6E	6D	6C	6B	6A	69	68
2CH	67	66	65	64	63	62	61	60
2BH	5F	5E	5D	5C	5B	5A	59	58
2AH	57	56	55	54	53	52	51	50
29H	4F	4E	4D	4C	4B	4A	49	48
28H	47	46	45	44	43	42	41	40
27H	3F	3E	3D	3C	3B	3A	39	38
26H	37	36	35	34	33	32	31	30
25H	2F	2E	2D	2C	2B	2A	29	28
24H	27	26	25	24	23	22	21	20
23H	1F	1E	1D	1C	1B	1A	19	18
22H	17	16	15	14	13	12	11	10
21H	0F	0E	0D	0C	0B	0A	09	08
20H	07	06	05	04	03	02	01	00
1FH	БАНК 3							R7
18H	. . . . .							R0
17H	БАНК 2							R7
10H	. . . . .							R0
0FH	БАНК 1							R7
08H	. . . . .							R0
07H	БАНК 0							R7
00	. . . . .							R0

	/(**IDATA**)/									
0F0H	F7	F6	F5	F4	F3	F2	F1	F0	B	для XA
0E0H	E7	E6	E5	E4	E3	E2	E1	E0	ACC	
	CV	AC	F0	RS1	RS0	0V		P		
0D0H	D7	D6	D5	D4	D3	D2	D1	D0	PSW	PSW51
			PT2	PS	PT1	PX1	PT0	PX0		
0B8H	-	-	BD	BC	BB	BA	B9	B8	IP	5 рег.
0B0H	B7	B6	B5	B4	B3	B2	B1	B0	P3	
	EA		ET2	ES	ET1	EX1	ET0	EX0		
0A8H	AF	-	AD	AC	AB	AA	A9	A8	IE	IEL
0A0H	A7	A6	A5	A4	A3	A2	A1	A0	P2	
	SM0	SM1	SM2	REN	TB8	RB8	TI	RI		
98H	9F	9E	9D	9C	9B	9A	99	98	SCON	S0CON S1CON
90H	97	96	95	94	93	92	91	90	P1	
	TF1	TR1	TF0	TR0	IE1	IT1	IE0	IT0		
88H	8F	8E	8D	8C	8B	8A	89	88	TCON	
80H	87	86	85	84	83	82	81	80	P0	

Регистры специальных функций: (карта адресов – лекции №1, №2). Перечень регистров специальных функций и всех символических имён собран в файлах заголовков для соответствующей модели МК: **reg51.h**, **reg52.h**. (На ассемблере – **reg51.inc**, **reg515.inc**, ….., **reg517.inc**, **reg52.inc**, **reg520.inc**, **reg528.inc**, ….. и т.д.)

КОМПИЛЯТОР C51 Keil Elektronik GmbH

Запуск компилятора с командной строки, синтаксис:

C51 <sourcefile> [<directives…>] или **C51 @<commfile>**

Например: C51 testfile.c SYMBOLS CODE DEBUG

#pragma SYMBOLS CODE DEBUG

1. Директивы и указания компилятору.

60 штук. Директивы, помеченные значком «†», могут быть определены только один раз в командной строке или в начале исходного текста в директиве #pragma. Они не могут использоваться больше чем один раз в исходном тексте программы, т.к относятся ко всему файлу.

Директивы можно разделить на **3 категории**: - управление исходным текстом; - управление листингом; - управление выдачей объектного файла.

Директивы для управления исходным текстом

DEFINE DF

Определяет символическое имя. Дополнительно может задать и значение имени. Допустима только в командной строке вызова компилятора, pragma игнорирует. (Аналог опций **-D**SYMB **-D**SYMB=xx для ICC8051.)

C51 SAMPLE.C DEFINE (check, NoExtRam)
C51 MYPROG.C DF (X1="1+5", iofunc="getkey ()")

INCDIR ID †

Определяет дополнительные пути поиска включаемых файлов (до 50 шт. через «;»), заданных в угловых скобках или кавычках. (Аналог опции **-I** <путь> для ICC8051.) Порядок поиска – «” … “» текущий каталог, каталог исход-ного файла, «< … >» данная директива, переменная окружения C51INC.

C51 SAMPLE.C INCDIR(F:\C51\MYINC;F:\CHIP_DIR)

NOEXTEND †

Компилятор работает в двух режимах. Эта директива запрещает Cx51 расширения стандарта ANSI Си, поддерживаемые компилятором по умолчанию при старте. (Аналог опции **-e** для ICC8051).

C51 SAMPLE.C NOEXTEND
#pragma NOEXTEND

Директивы управления листингом компиляции

CODE CD †

Добавляет ассемблерные строки соответствующих выражений текста Си в файл выходного листинга. Полезна для анализа компиляции конструкций Си в команды ассемблера.

C51 SAMPLE.C CD
#pragma code

COND CO,
NOCOND NOCO †

Включает или исключает исходные линии текста, входящие в ложные условные блоки компиляции.

EJECT EJ

Вставляет символ перевода страницы для листинга, продолжающегося с новой страницы.

#pragma eject

LISTINCLUDE LC

Содержимое включаемых файлов заносится в файл листинга.

C51 SAMPLE.C LISTINCLUDE
#pragma listinclude

PAGELENGTH PL †

Определяет число строк в выходном листинге на странице. По умолчанию их 60.

C51 SAMPLE.C PAGELENGTH (70)
#pragma pl (70)

PAGEWIDTH PW †

Определяет максимальное число символов в строке листинга.

C51 SAMPLE.C PAGEWIDTH(79)
#pragma pw(79)

PREPRINT PP †

Производит выдачу листинга препроцессора, где все макросы расширены. Только в командной строке задаётся, без директивы pragma.

C51 SAMPLE.C PREPRINT
C51 SAMPLE.C PP (PREPRO.LSI)

PRINT PR,
NOPRINT NOPR †

Определяет название(имя) для файла листинга, или запрещает листинг.

C51 SAMPLE.C PRINT(CON:)
#pragma pr (\usr\list\sample.lst)
C51 SAMPLE.C NOPRINT
#pragma nopr

SYMBOLS SB †

Включает список всех символов, используемых в пределах программного модуля в файле листинга.

C51 SAMPLE.C SYMBOLS
#pragma SYMBOLS

WARNINGLEVEL WL †

Задаёт уровень выдачи предупреждающих сообщений от 0 до 2. 0 - не выдавать; 2 - выдавать всё (по умолчанию); 1 - только то, что может дать некорректный код.

C51 SAMPLE.C WL (1)
#pragma WARNINGLEVEL (0)

Директивы управления созданием объектного файла

AREGS AR,
NOAREGS NOAR

Позволяет или запрещает абсолютную адресацию регистров. Задаются только вне определений функций. Может переключать режим несколько раз в программе. Укорачивает код записи в стек, минуя аккумулятор. Задаётся столько раз в программе, сколько нужно.

```
#pragma NOAREGS
noaregfunc() {
    k = func() + func();
}
#pragma AREGS
aregfunc() {
    k = func() + func();
}
```

ASM,
ENDASM

Включает фрагменты текста программы на языке ассемблера в текст программы на языке Си. Отмечает начало и конец встроенного блока текста ассемблера. Используется только в тексте прграммы в директиве #pragma.

```
main () {
 test ();
 ...
 #pragma asm                                                    другой вариант!
 JMP $  ; endless loop          ; SJMP $                        __asm sjmp $
 #pragma endasm
}
```

Есть специальная директива для получения ассемблерного текста всей программы (SRC).

BROWSE **BR †**

Позволяет создавать компилятору информацию о символах программы для интегрированной среды.

```
C51 SAMPLE.C BROWSE
#pragma browse
```

COMPACT **CP †**

Задаёт КОМПАКТНУЮ модель памяти. ПО умолчанию переменные и параметры размещаются в памяти PDATA (256 байт, младшая страница).

```
C51 SAMPLE.C COMPACT
#pragma compact
```

DEBUG **DB †**

Включает информацию для отладки в объектный файл.

```
C51 SAMPLE.C DEBUG
#pragma db
```

DISABLE

Запрещает прерывания для функций. Задаётся в директиве #pragma и перед определяемой функцией. (Тип функций monitor для ICC8051.)

```
#pragma disable /* Disable Interrupts */
uchar dfunc (uchar p1, uchar p2) {
 return (p1 * p2 + p2 * p1);
}
```

FLOATFUZZY **FF**

Определяет число знаков для округления чисел с плавающей запятой в операциях сравнения.

```
C51 MYFILE.C FLOATFUZZY (2)
#pragma FF (0)
```

INTERVAL †

Задаёт интервал для создания таблицы векторов прерываний. Используется формула (interval * n) + offset + 3. Здесь нужно **8**. (Заметим, что для архитектуры XA кратность равна **4.**)

```
C51 SAMPLE.C INTERVAL(3)
#pragma interval(3)
```

INTPROMOTE **IP,**
NOINTPROMOTE **NOIP †**

Разрешает или запрещает ANSI расширение коротких целых данных в более длинные целые числа. Важно: как выполнять расширение знака в выражениях.

```
C51 SAMPLE.C INTPROMOTE
#pragma intpromote
C51 SAMPLE.C NOINTPROMOTE
```

INTVECTOR **IV,**
NOINTVECTOR **NOIV †**

Определяет смещение таблицы векторов прерываний или запрещает генерацию таблицы векторов.

```
C51 SAMPLE.C INTVECTOR(0x8000)
#pragma iv(0x8000)
C51 SAMPLE.C NOINTVECTOR
#pragma noiv
```

LARGE LA †

Задаёт БОЛЬШУЮ модель памяти. Все переменные по умолчанию в памяти XDATA.

```
C51 SAMPLE.C LARGE
#pragma large
```

MAXARGS †

Определяет максимальный размер блоков списков аргументов для функций.

```
C51 SAMPLE.C MAXARGS(20)
#pragma maxaregs (4)                /* allow 4 bytes for parameters */
```

MOD517,
NOMOD517

Позволяет или запрещает коды, специфичные для дополнительных особенностей аппаратных средств ЭВМ 80C517 и его производных (Infineon C517).

```
C51 SAMPL517.C MOD517
#pragma MOD517 (NOAU)
#pragma MOD517 (NODP8)
#pragma MOD517 (NODP8, NOAU)
C51 SAMPL517.C NOMOD517
#pragma NOMOD517
```

MODA2,
NOMODA2

Позволяет или запрещает поддержку двух регистров DPTR для Atmel 82x8252 и его вариантов.

```
C51 SAMPLE.C MODA2
#pragma moda2
C51 SAMPLE.C NOMODA2
#pragma nomoda2
```

MODDP2,
NOMODDP2

Позволяет или запрещает поддержку двух регистров DPTR для МК Dallas 80C320, C520, C530, C550.

```
C51 SAMPL320.C MODDP2
#pragma moddp2
C51 SAMPL320.C NOMODDP2
#pragma nomoddp2
```

MODP2,
NOMODP2

Позволяет или запрещает поддержку двух регистров DPTR для Philips и Temic и их производных.

```
C51 SAMPLE.C MODP2
#pragma modp2
C51 SAMPLE.C NOMODP2
#pragma nomodp2
```

NOAMAKE NOAM †

Запрещает генерировать информацию AutoMAKE, нужную для сторонних средств разработки. По умолчанию генерируется.

```
C51 SAMPLE.C NOAMAKE
#pragma NOAM
```

OBJECT OJ,
NOOBJECT NOOJ †

Позволяет задать объектный файл, произвольно определить название, или подавите файл объекта.

```
C51 SAMPLE.C OBJECT(sample1.obj)
#pragma oj(sample_1.obj)
C51 SAMPLE.C NOOBJECT
#pragma nooj
```

OBJECTEXTEND OE †

Включает дополнительную информацию о типе переменных в объектный файл для символьной отладки симуляторами.

```
C51 SAMPLE.C OBJECTEXTEND DEBUG
#pragma oe db
```

ONEREGBANK OB

Задаёт, что только 0 банк регистров используется в коде программ прерывания. Сокращает код, не используя управление для PSW.

```
C51 SAMPLE.C ONEREGBANK
#pragma OB
```

OMF2 O2 †

Производит расширенный формат выходного файла OMF2 (расширение OMF51). (Выходных форматов уже

насчитывается несколько десятков, начиная от самых простых первых.)

```
C51 SAMPLE.C OMF2
#pragma O2
```

OPTIMIZE OT

Определяет уровень оптимизации, выполняемой компилятором. От 0 до 9 для кода или скорости.

```
C51 SAMPLE.C OPTIMIZE (9)
C51 SAMPLE.C OPTIMIZE (0)
#pragma ot(6, SIZE)
#pragma ot(size)
```

ORDER OR †

Переменные размещаются в порядке, в котором они появляются в исходном файле.

```
C51 SAMPLE.C ORDER
#pragma OR
```

REGFILE RF †

Определите файл определений регистров для глобальной оптимизации регистров.

```
C51 SAMPLE.C REGFILE(sample.reg)
#pragma REGFILE(sample.reg)
```

REGISTERBANK RB

Выберает банк регистров, который используется для последующих функций. От 0 до 3. Только объявляется.

```
C51 SAMPLE.C REGISTERBANK(1)
#pragma rb(3)
```

REGPARMS,
NOREGPARMS

Разрешает или запрещает получать функциям три аргумента в регистрах. Код короче и быстрее. Для совместимости со старыми библиотеками запрещаются аргументы в регистрах.

```
C51 SAMPLE.C NOREGPARMS
#pragma NOREGPARMS /* Parm passing-old method */
extern int old_func (int, char);
#pragma REGPARMS /* Parm passing-new method */
extern int new_func (int, char);
```

RET_PSTK RP †,
RET_XSTK RX †

Разрешают использование внешних стеков в PDATA и в XDATA в вызовах функций. Полезна для реентерантных процедур. Надо модифицировать стартовый код STARTUP.51 .

```
C51 SAMPLE.C RET_XSTK
#pragma RET_PSTK
extern void func2 (void);
void func (void) {
  func2 ();
}
```

ROM †

Специфицирует размер программной памяти. Управляет вызовами и переходами - AJMP/ACALL или LJMP/LCALL инструкции. (SMALL до 2 Кб AJMP и ACALL только; COMPACT – AJMP и LCALL; LARGE – все далёкие; для Dallas 390 D512K – AJMP и ACALL с 19-битным адресом, и D16M – LCALL с 24-битным адресом и AJMP с 19-битным адресом.)

```
C51 SAMPLE.C ROM (SMALL)
#pragma ROM (SMALL)
```

SAVE,
RESTORE

Используется только в исходном тексте программы. Сохраняет и восстанавливает текущие назначения для AREGS, REGPARMS и OPTIMIZE директив.

```
#pragma save
#pragma noregparms
extern void test1 (char c, int i);
extern char test2 (long l, float f);
#pragma restore
```

SMALL SM †

Задаёт МАЛЕНЬКУЮ модель памяти. (По умолчанию).

```
C51 SAMPLE.C SMALL
#pragma small
```

SRC †

Создавать исходный файл на языке ассемблера вместо объектного модуля. Затем его можно ассемблировать.

```
C51 SAMPLE.C SRC
```

C51 SAMPLE.C SRC(SML.A51)

STRING ST †

Расположить неявные строки-константы в XDATA или далекой памяти.

C51 SAMPLE.C STRING (XDATA)

#pragma STRING (FAR)

VARBANKING VB †

Использовать библиотеки модели банков памяти для переменных во внешней памяти.

C51 SAMPLE.C VARBANKING

#pragma VARBANKING

2. Расширение стандарта ANSI (основной режим), ключевые слова.

_at_	idata	sfr
alien	interrupt	sfr16
bdata	large	small
bit	pdata	_task_
code	_priority_	using
compact	reentrant	xdata
data	sbit	

Битовые данные – **bit,** 128 бит внутренней памяти данных. Тип памяти программ – **code**. Тип внутренней памяти данных – **data**, 128 байт; **idata**, 256 байт; **bdata**, 16 байт от 20h до 2Fh, допускают доступ как к байтам, словам, двойным словам, так и к битам. Тип внешней памяти данных – **pdata**, младшие 256 байт; **xdata**, 64 Кбайт. Регистры специальных функций – **sfr,** 8-разрядные регистры; **sfr16**, 16-разрядные регистры из внутренней памяти от 80h до 0FFh. Битовая адресация – **sbit**, для переменных **bdata, sfr, sfr16**. Абсолютная адресация переменных – модификатор **_at_**.

Остальные слова – атрибуты и модификаторы для определения функций (и модели памяти).

3. Модели памяти.

Модели памяти неявно задают тип памяти переменных, аргументов и возвращаемых значений функций, которые можно изменить явным указанием типа памяти.

Малая модель – **small** – по умолчанию все данные во внутренней памяти данных **data**. Обеспечивает быстрый доступ и короткий код. (Аналогична tiny для ICC8051.)

Компактная модель – **compact** – по умолчанию все данные в первой странице внешней памяти данных **pdata.** Аналогий с ICC8051 нет.

Большая модель – **large** – по умолчанию все данные во внешней памяти данных **xdata**. Достаточно громоздкая и медленная. (Аналогична large для ICC8051.)

В рамках данных моделей поддерживаются стандартные **типы данных**: скалярные и комбинируемые в структуры, объединения и массивы. Символом † отмечены специфические для семейства MCS-51 типы.

Тип данных	биты	байты	диапазон значений
bit †	1		от 0 до 1
signed char	8	1	от -128 до +127
unsigned char	8	1	от 0 до 255
enum	16	2	от -32768 до +32767
signed short	16	2	от -32768 до +32767
unsigned short	16	2	от -0 до 65535
signed int	16	2	от -32768 до +32767
unsigned int	16	2	0 до 65535
signed long	32	4	от -2147483648 до 2147483647
unsigned long	32	4	от 0 до 4294967295
float	32	4	от ±1.175494E-38 до ±3.402823E+38
sbit †	1		от 0 до 1
sfr †	8	1	от 0 до 255
sfr16 †	16	2	от 0 до 65535

Битовые типы всегда имеют свой прямой адрес, поэтому не указуемы.

Остальные типы могут иметь **указатели** в привычном объявлении.

Традиционно, типы **double** и **long double** транслируются в числа с плавающей запятой одинарной точности.

4. Явное объявление типа памяти.

Используется суффиксная нотация написания модификаторов в декларациях и определениях. Допустима и префиксная нотация (как у ICC8051) для совместимости с другими компиляторами. Примеры:

char **data** var1;	**data** char var1;
char code text[] = "ENTER PARAMETER:";	code char text[] = "ENTER PARAMETER:";
unsigned long xdata array[100];	xdata unsigned long array[100];
float idata x,y,z;	idata float x,y,z;
unsigned int pdata dimension;	pdata unsigned int dimension;
unsigned char xdata vector[10][4][4];	xdata unsigned char vector[10][4][4];
char bdata flags;	bdata char flags;

5. Бит-адресуемые объекты

– это все целочисленные объекты (char, short, int, long; signed и unsigned), допускающие доступ одновременно и

как к байтам, и как к битам (смотри начало лекции). Память для этого класса данных – небольшая, всего 16 байт памяти **bdata**. Примеры:

```
int bdata ibase;          /* Bit-addressable int */
char bdata bary[4];       /* Bit-addressable array */
```

Доступ к битам этих объектов осуществляется посредством объявления переменных типа **sbit**. В данном случае определяются только имена переменных, память для них резервируется раньше. Синтаксис такого объявления следующий:

sbit <имя> = <базовая переменная> ^ <номер бита>;
sbit <имя> = <конст. базовый адрес> ^ <номер бита>;
sbit <имя> = <конст. адрес>;

```
sbit mybit0 = ibase ^ 0;      /* bit 0 of ibase */
sbit mybit15 = ibase ^ 15;    /* bit 15 of ibase */
sbit Ary07 = bary[0] ^ 7;     /* bit 7 of bary[0] */
sbit Ary37 = bary[3] ^ 7;     /* bit 7 of bary[3] */
```

```
Ary37 = 0;       /* clear bit 7 in bary[3] */
bary[3] = 'a';   /* Byte addressing */
ibase = -1;      /* Word addressing */
mybit15 = 1;     /* set bit 15 in ibase */
```

Тип **sbit** ведёт физическую адресацию бит на основании своего базового типа. Для слов и двойных слов здесь есть разница с логической адресацией бит из-за перестановки байт. Для типа **int** пример:

```
 0 1  2   3    4   5  6   7  8 9 10 11 12 13 14 15 16 …      физические адреса бит (последовательность бит)
|   байт 0    ст          |   байт 1     мл        |        байты в памяти
 8 9 10 11 12 13 14 15 0 1  2   3   4   5   6   7           логические адреса бит
```

Переменные типа **sbit** допустимы лишь для данных **bdata, sfr** и **sfr16,** которые имеют битовую организацию. Пример 3 вариантов задания переменных из имеющихся данных:

```
sfr16 T2 = 0xCC;         /* Timer 2: T2L 0CCh, T2H 0CDh */
sfr16 RCAP2 = 0xCA;      /* RCAP2L 0CAh, RCAP2H 0CBh */
```

```
sfr PSW = 0xD0;
sfr IE = 0xA8;
sbit OV = PSW ^ 2;
sbit CY = PSW ^ 7;
sbit EA = IE ^ 7;
```

```
sbit OV = 0xD0 ^ 2;
sbit CY = 0xD0 ^ 7;
sbit EA = 0xA8 ^ 7;
sbit T2b4 = 0xCC ^ 4;
```

```
sbit OV = 0xD2;
sbit CY = 0xD7;
sbit EA = 0xAF;
```

6. Данные с абсолютными адресами.

Эти данные не инициализируются при объявлении. (no_init для ICC8051.) Функции и переменные типа бит этим модификатором не пользуются, т.к. их адреса уже заданы. Синтаксис задания адреса (префиксная нотация типа памяти):

[<тип памяти(н)>] <тип> <имя> _at_ <константа адреса>

Пример:

```
idata struct link list _at_ 0x40;        /* list at idata 0x40 */
xdata char text[256] _at_ 0xE000;        /* array at xdata 0xE000 */
xdata int i1 _at_ 0x8000;                /* int at xdata 0x8000 */
```

7. Указатели данных.

Два типа указателей – общие и типизированные. Общие занимают 3 байта памяти. Типизированные короче: 1 байт для типов **data, idata, bdata** и **pdata**; 2 байта для типов **code** и **xdata**.

Общие указатели. В объявлении типа нет спецификатора памяти. Поле типа указателя (отличается от ICC8051) имеет следующие значения: 0 – для **data** и **idata**, 1 – для **xdata**, -1 – для **code**, -2 – для **pdata** . Общие указатели могут размещаться в памяти по умолчанию, а можно и явно задать их размещение с помощью модификаторов типа памяти:

По умолчанию задаются моделью памяти:

```
char *c_ptr;      /* char ptr */
int *i_ptr;       /* int ptr */
long *l_ptr;      /* long ptr */
char data dj;     /* data vars */
int xdata xk;
long code cl = 123456789;
c_ptr = &dj;      /* data ptrs */
i_ptr = &xk;      /* xdata ptrs */
l_ptr = &cl;      /* code ptrs */
```

Явное размещение:

```
char * xdata strptr;  /* generic ptr stored in xdata */
int * data numptr;    /* generic ptr stored in data */
long * idata varptr; /* generic ptr stored in idata */
```

Типизированные указатели. Присутствует спецификатор памяти в объявлении типа объекта. Использование их сокращает программный код и размер данных под них. Размещение их аналогичное общим.

По умолчанию задаются моделью памяти:

```
char data *str;          /* ptr to string in data */
int xdata *numtab;       /* ptr to int(s) in xdata */
long code *powtab;       /* ptr to long(s) in code */
```

Явное размещение:

```
char data * xdata str;        /* ptr in xdata to data char */
int xdata * data numtab;      /* ptr in data to xdata int */
long code * idata powtab;     /* ptr in idata to code long */
```

Преобразования указателей общих в типизированные или обратно осуществляет компилятор в соответствии с их форматами.

Допустимы и костантные типизированные указатели. Например: pc = (char code *) 0x1342 .

8. Функции.

Функция - это независимая совокупность объявлений и операторов, обычно предназначенная для выполнения определенной задачи. Программы на Си состоят по крайней мере из одной функции **main**, но могут содержать и больше функций. Описываются, определяются, объявляются и вызываются функции Си стандартным образом. Компилятор C51 полностью соответствует стандарту ANSI и дополнительно поддерживает атрибуты функций, специфичные для MCS-51.

8.1. Общие положения.

Определение функции специфицирует имя функции, её формальные параметры, объявления и операторы, которые определяют её действия. В определении функции может быть задан также тип возврата и её класс памяти.

В объявлении задается имя, тип возврата и класс памяти функции, чьё явное определение произведено в другой части программы. В объявлении функции могут быть также специфицированы число и типы аргументов функции. Это позволяет компилятору сравнить типы действительных аргументов и формальных параметров функции. Объявления не обязательны для функций, возвращающих величины типа int. Чтобы обеспечить корректное обращение при других типах возвратов, необходимо объявить функцию перед её вызовом.

Вызов функции передает управление из вызывающей функции к вызванной. Действительные аргументы, если они есть, передаются по значению в вызванную функцию. При выполнении оператора return в вызванной функции управление и, возможно, значение возврата передаются в вызывающую функцию.

8.1.1. Определение функции.

Определение функции специфицирует имя, формальные параметры и тело функции. Оно может также определять тип возврата и класс памяти функции. Синтаксис определения обычной функции следующий:

[<sc-specifier>][<type-specifier>]<declarator>([<parameter-list>])
[**small** | **compact** | **large**][**reentrant**][**interrupt** n][**using** n] порядок важен
[<parameter-declarations>] { K & R
<function-body> }

Спецификатор класса памяти <sc-specifier> задает класс памяти функции, который может быть или **static** или **extern**. Спецификатор типа <type-specifier> и декларатор <declaration> специфицируют тип возврата и имя функции. Список параметров <parameter-list> - это список (возможно пустой) формальных параметров, которые используются функцией. Объявления параметров <parameter-declarations> задают типы формальных параметров в старой нотации (для совместимости). Тело функции <function-body> - это составной оператор, содержащий объявления локальных переменных и операторы.

В определении функции могут присутствовать ещё атрибуты: **alien** (префиксный), **_task_** и **_priority_.**

8.1.2. Класс памяти

Спецификатор класса памяти в определении функции определяет функцию как static или extern. Функция с классом памяти **static** видима только в том исходном файле, в котором она определена. Все другие функции с классом памяти **extern**, заданным явно или неявно, видимы во всех исходных файлах, которые образуют программу.

Если спецификатор класса памяти опускается в определении функции, то подразумевается класс памяти extern. Спецификатор класса памяти extern может быть явно задан в определении функции, но этого не требуется. Extern может употребляться в программе много раз в объявлениях.

Спецификатор класса памяти требуется при определении функции только в одном случае, когда функция объявляется где-нибудь в другом месте в том же самом исходном файле с спецификатором класса памяти static. Спецификатор класса памяти static может быть также использован, когда определяемая функция предварительно объявлена в том же самом исходном файле без спецификатора класса памяти. Как правило, функция, объявленная без спецификатора класса памяти, подразумевает класс extern. Однако, если определение функции явно специфицирует класс static, то функции даётся класс static.

8.1.3. Тип возврата

Тип возврата функции определяет размер и тип возвращаемого значения. Объявление типа имеет следующий синтаксис:

[<type-specifier>] <declarator>,

где спецификатор типа <type-specifier> вместе с декларатором <declarator> определяет тип возврата и имя функции. Если <type-specifier> не задан, то подразумевается, что тип возврата **int**. Спецификатор типа может специфицировать основной, структурный и совмещающий типы. Декларатор состоит из идентификатора функции, возможно модифицированного с целью объявления адресного типа. Функции не могут возвращать массивов или функций, но они могут возвращать указатели на любой тип, включая массивы и функции. Тип возврата, задаваемый в определении функции, должен соответствовать типам возвратов, заданных в объявлениях этой функции, сделанных где-то в программе. Функции с типом возврата int могут не объявляться перед вызовом. Функции с другими типами возвратов не могут быть вызваны прежде, чем они будут определены или объявлены.

Тип значения возврата функции используется только тогда, когда функция возвращает значение, которое вырабатывается, если выполняется оператор return, содержащий выражение. Выражение вычисляется, преобразуется к типу возврата, если это необходимо, и возвращается в точку вызова. Если оператор return не выполняется или если выполняемый оператор return не содержит выражения, то значение возврата функции не определено. Если в этом случае вызывающая функция ожидает значение возврата, то поведение программы также не определено.

Пример:
```
/* return type is int */
static add (x,y)
int x,y              // старая нотация K&R
{
  return (x+y);
}
```

8.1.4. Аргументы и локальные переменные

Аргументы и локальные переменные функций размещаются статически в фиксированной памяти в соответствии с заданной моделью памяти (**small, compact, large**). Только адрес возврата записывается в стек при вызове и извлекается оттуда при возврате. Такой подход работает при малом стеке.

8.1.5. Передача параметров в регистрах

Компилятор разрешает до 3 аргументов функции передавать в регистрах. Это улучшает производительность программы. Эта возможность разрешается директивой **REGPARMS** (отменяется **NOREGPARMS**).

Номер аргумента	char, 1-байт ptr	int, 2-байт ptr	long, float	общий ptr
1	R7	R6 & R7	R4—R7	R1—R3
2	R5	R4 & R5	R4—R7	R1—R3
3	R3	R2 & R3		R1—R3

Параметры желательно располагать по этой схеме. Тип **bit** лучше не объявлять среди первых трёх, если набирается столько. Если первый параметр типа **bit**, то остальные параметры не передаются в регистрах. Пример:

```
int function (
  int v_a,          /* размещается в R6 и R7 */
  char v_b,         /* размещается в R5 */
  bit v_c,          /* размещается в фиксированной памяти, т.к. бит */
  long v_d,         /* размещается в фиксированной памяти, т.к. 4-ый */
  bit v_e) {        /* размещается в фиксированной памяти, т.к. 5-ый */
   … // тело функции
}
```

Возвращаемое значение всегда передаётся в жёстко закреплённых регистрах (как для ICC8051).

Возвращаемый тип	регистры	описание
bit	Carry Flag	
char, **unsigned char, 1-byte pointer**	R7	
int, **unsigned int, 2-byte pointer**	R6 & R7	Регистровая пара MSB в R6, LSB в R7
long, **unsigned long**	R4 — R7	Регистровая тетрада MSB в R4, LSB в R7
float	R4 — R7	32-Bit IEEE format
общий указатель	R1 — R3	Memory type в R3, MSB в R2, LSB в R1

8.1.6. Модели памяти

Аргументы и локальные переменные функций неявно располагаются в соответствии с моделью памяти. Эту договорённость можно изменить явным заданием модели:

```
#pragma small                   /* Default to small model */
extern int calc (char i, int b) large reentrant;
extern int func (int i, float f) large;
extern void *tcp (char xdata *xp, int ndx) small;
int mtest (int i, int y)        /* Small model */
{
  return (i *y +y *i +func(-1, 4.75));
}
```

8.1.7. Использование другого банка регистров

По умолчанию задаётся 0 банк регистров. Явное задание банка регистров – ключевое слово **using** (для ICC8051 в этой ситуации использовались квадратные скобки.):

```
void rb_function (void) using 3 {
  .
  .
}
```

Аргумент – константа от 0 до 3. Компилятор генерирует код переключения банков самостоятельно. Применимо к любым функциям, а не только для обработчиков прерываний.

Ограничения. (1) Не используется, если функция возвращает значение в регистрах. (2) Нельзя, если возвращаемое значение из функции типа **bit**.

8.1.8. Банк регистров по умолчанию

По сигналу аппаратного сброса всегда устанавливается 0 банк регистров для основной работы. Банк регистров по умолчанию можно переобъявить директивой **REGISTERBANK**. Эта директива не генерирует код для переключения банка регистров, а управляет абсолютной адресацией регистров. **Важно,** программист сам модифицирует код **STARTUP.A51** для переключения банка, и использует директиву REGISTERBANK (n).

```
MOV    PSW, #018H       ; установить банк 3
```

```
…        /* код */
#pragma rb(3)              // в своей программе
```

По умолчанию компилятор использует абсолютную адресацию регистров для укорочения кода и ускорения его исполнения. Тогда функции не должны вызывать другие функции с отличными банками регистров. Чтобы устранить эту ситуацию и сделать функции не чувствительными к перемене банков, используется директива **NOAREGS**.

8.1.9. Реентерантные функции

Базовая архитектура 8051 не поддерживает реентерантные процедуры из-за короткого физического стека. Однако решить данную проблему можно соответствующими программными средствами. Сколько раз вызывается процедура, столько раз копии её параметров и локальных переменных должны присутствовать в памяти. Для этого создаются специальные реентерантные стеки. В данном случае для каждого типа памяти (модели памяти) создаётся свой реентерантный стек, который растёт в сторону уменьшения адресов. **Важно,** (1) программист сам модифицирует код **STARTUP.A51** для задания нужных ему реентерантных стеков. Кроме того, (2) все реентерантные процедуры программист помечает специальным атрибутом **reentrant**. Здесь могут быть три реентерантных стека: для малой модели (**small**) памяти в **idata**, для компактной (**compact**) – в **pdata**, для большой (**large**) – в **xdata**. Какой тип функций присутствует в программе, такие стеки и нужно задать. Пример:

```
int calc (char i, int b) reentrant {          // модель памяти по умолчанию small
  int x;
  x = table [i];                                          ( 1 + 2 + 2 = 5,   2 для x)
  return (x * b);
}
```

Компилятор генерирует код так, что только адрес возврата хранится в стеке, а все остальные параметры и локальные переменные – в соответствующем реентерантном стеке. Используются следующие указатели реентерантных стеков:

Модель	Указатель стека	Область стека
SMALL	?C_IBP (1 байт)	внутренняя память (idata), 256 байт максимум.
COMPACT	?C_PBP (1 байт)	внешняя память (pdata) , 256 байт максимум.
LARGE	?C_XBP (2 байта)	внешняя память (xdata). 64 Кбайт.

Ограничения:

(1). Реентерантные функции не должны иметь аргументов типа **bit**, т.к. их память фиксирована;
(2). Реентерантные функции не должны вызывать чужие функции с атрибутом **alien**;
(3). Реентерантные функции не должны сами иметь дополнительный атрибут **alien**;
(4). Реентерантные функции могут иметь дополнительно лишь атрибуты модели памяти;
(5). Реентерантные функции могут иметь дополнительно лишь атрибуты **using, interrupt**;
(6). Реентерантные функции имеют атрибут **reentrant** только в определении, (в прототипе раньше нельзя было).

8.1.10. Обработчики прерываний

Чтобы определить обработчик прерывания, используется атрибут **interrupt**. Базовая архитектура 8051 предусматривает 5 прерываний с адресами через 8 байт:

Номер прерывания	Описание	Адрес вектора
0	EXTERNAL INT 0	0003h
1	TIMER/COUNTER 0	000Bh
2	EXTERNAL INT 1	0013h
3	TIMER/COUNTER 1	001Bh
4	SERIAL PORT	0023h

Синтаксис определения функции-обработчика прерывания предусматривает задание не адреса вектора прерывания, а только его номера. В системе зарезервировано 32 номера (на всю внутреннюю память), которые указываются константами от 0 до 31. Пример (таймер отсчитывает секунды) (для ICC8051 в этой ситуации использовались квадратные скобки.):

```
unsigned int interruptcnt;
unsigned char second;
 void timer0 (void) interrupt 1 using 2 {
   if (++interruptcnt == 4000) {          /* count to 4000 */
     second++;                            /* second counter */
     interruptcnt = 0;                    /* clear int counter */
   }
 }
```

При входе в процедуру сохраняются **ACC, B, DPH, DPL** и **PSW** и рабочие регистры, что используются, а перед выходом этот контекст восстанавливается и выполняется **RETI**.

Ограничения:

(1). Функции прерывания не должны иметь атрибут **interrupt** в прототипе, а только в определении, т.к. в теле функции сохраняются рабочие регистры при входе и восстанавливаются при выходе, выход RETI (см. ранее);
(2). Функции прерывания не должны иметь аргументов, т.к. неоткуда их взять – (void);
(3). Функции прерывания не должны возвращать результат, т.к. его некому присвоить – void;

(4). Функции прерывания нельзя явно вызывать в программе, компилятор это отслеживает;

(5). Косвенные вызовы компилятор не отследит, но делать их не вижу смысла;

(6). Компилятор генерирует вектор прерывания с далёким переходом самостоятельно. Это можно отменить директивой **NOINTVECTOR**, тогда вектор придётся устанавливать самому в другом модуле;

(7). Номер прерывания задаётся константой;

(8). Функция прерывания допускает использование атрибута **using;**

(9). Функция прерывания может вызывать другую функцию с тем же банком регистров, что и сама. Ситуация как с директивой **NOAREGS.**

8.1.11. Функции PL/M-51

Есть другой интерфейс вызова процедур: в частности возвращаемые байтовые значения передаются через аккумулятор; регистры в передаче параметров не участвуют. Это свойственно достаточно популярному языку PL/M-51 для МК-систем наряду с Си (формульный, родственный). Интерфейс функций PL/M-51 задаётся ключевым словом **alien**. Компилятор позволяет вызывать эти функции и генерировать код для них. Аргументы этих функций и возвращаемое значение могут быть лишь типа **bit, char, unsigned char, int,** и **unsigned int**, что гарантирует компилятор. Другие типы задавать можно, правда без гарантии правильной работы сними. **Недопустимо** объявлять в функции переменное число параметров.

Использование

```
extern alien char plm_func (int, char);
char c_func (void) {
  int i;
  char c;
  for (i = 0; i < 100; i++) {
    c = plm_func (i, c);      /* call PL/M func */
  }
  return (c);
}
```

Определение

```
alien char c_func (char a, int b) {
  return (a * b);
}



// ошибка!
extern alien unsigned int plm_i (char, int, ... );
```

8.1.12 Функции для многозадачной системы реального времени

Каждая программа выполняется как отдельная законченная задача. Квазипараллельное выполнение задач на одном процессоре в режиме разделения времени. Для каждой задачи выделяется свой интервал времени. Все задачи последовательно ставятся на выполнение. На большом промежутке времени сказывается иллюзия, что задачи выполняются параллельно. Это становится более правдоподобным, если часть задач ожидают внешних сообщений. Синхронизовать несколько задач можно посредством обмена сообщениями между задачами.

Для отсчёта временных интервалов в системе используются аппартные таймеры на кристалле: для крошечной модели T0, для полной – оба таймера T0 и T1.

Функции-задачи не имеют аргументов и не возвращаю результатов своей работы.

Ключевое слово **_task_** служит для задания ID номера функции при использовании её как задачи реального времени. Ключевое слово **_priority_**.задаёт приоритет задачи.

```
void job0(void) _task_ num _priority_ pri
```

Для крошечной системы RTX51 Tiny идентификаторы задач от 0 до 15, т.е. поддерживает максимум 16 задач. Для полной системы RTX51 Full – они от 0 до 255, т.е. поддерживает максимум 256 задач, 19 из них могут быть активны. Полная система поддерживает ещё и механизм приоритетов задач.

9. ПРЕПРОЦЕССОР Си

Препроцессор Си - это текстовый процессор, используемый для обработки текстового исходного файла на первой фазе компиляции. Используется для макроподстановки, условной компиляции, включения именованных файлов и т.д.. Строки, начинающиеся с знака **#** (перед ним возможны лишь пробельные символы), устанавливают связь с препроцессором. Компилятор обычно вызывает препроцессор в своем первом проходе, однако препроцессор может быть вызван автономно для обработки текста без компиляции. Работа:

1. - выбрасываются пары литер, состоящие из «\» с последующей литерой «новая строка»; т.е. осуществляется склеивание строк.

2. – программа разбивается на лексемы, разделённые пробельными литерами. Комментарии заменяются на единичные пробелы. Затем выполняются директивы препроцессора и макроподстановки.

3. – Ecs-последовательности в строковых и символьных константах заменяются на литеры, которые они обозначают. Соседние строковые литералы соединяются.

4. – Результат транслируется.

9.1. Директивы препроцессора

Директивы препроцессора - это инструкции, предназначенные для препроцессора Си. Директивы препроцессора обычно используются для того, чтобы сделать исходные программы проще для модификации и чтобы осуществлять компиляцию для различных реализаций компилятора Си. Директивы в исходном файле инструктируют препроцессор о выполнении определенных действий. Например, препроцессор может заменить лексемы в тексте, вставить содержимое других файлов в исходный файл, запретить компиляцию части файла и т.д.

Препроцессор Си распознает по ANSI-стандарту следующие 13 директив:

# define	**# if**	**# line**	**#**
# elif	**# ifdef**	**# undef**	

 # else **# ifndef** **# error**
 # endif **# include** **# pragma** **(#message)**

Знак номера (#) должен быть первым (без предшествующих пробельных символов) в строке, содержащей директиву. Пробельные символы допускаются между знаком номера и первой буквой директивы. Некоторые директивы требуют аргументы или значения. Директивы могут появляться где угодно в исходном файле, но они применимы только к остатку исходного файла от места, где они появились.

ICC8051 имеет дополнительную директиву **# message "<текст>"** для выдачи предупреждений.

9.1.1. Поименованные константы и макросы

Директива **# define** обычно используется для связи мнемоничных идентификаторов (псевдонимов) с константами, ключевыми словами, операторами и выражениями, которые часто используются. Идендтификаторы, которые представляют константы, называются поименованными константами. Идентификаторы, которые представляют операторы или выражения, называются макросами.

Идентификатор не может быть переопределен без отмены первого определения. Однако, идентификатор может быть переопределен точно таким же определением. Таким образом в программе допускается повторение одного и того же определения.

Директива **# undef** отменяет определение идентификатора. Когда определение отменено, то идентификатор может быть сопоставлен с другим значением. Макросы могут быть определены по образу и подобию с вызовами функций. Поскольку макросы не вырабатывают действительных вызовов функций, то замена вызовов функций макросами может повысить скорость выполнения программы. Однако макросы создают проблемы, если они тщательно не определены. Макро-определения с аргументами могут потребовать использования круглых скобок для определения старшинства операций в выражениях. Макросы могут некорректно повлиять на выражения с побочными эфектами.

Директива # define , синтаксис:

 # define <identifier><text> псевдоним
 # define <identifier>(<parameter-list>)<text> макрос

Директива # define заменяет все вхождения идентификатора <identifier> в исходной программе на <text>**.** Идентификатор заменяется, если он оформлен в виде лексемы. Например, идентификатор не изменяется, если он представлен внутри строки или как часть более длинного идентификатора. Если после идентификатора следует список параметров <parameter-list>, то директива # define заменяет каждое вхождение выражения <identifier (parameter-list) на <text>, модифицированный заменой формальных аргументов фактическими.

<text> представляет собой набор лексем, таких как ключевые слова, константы или составные операторы. Один или более пробельных символов могут разделять <text> от <identifier> (или от заключенных в скобки параметров). Если текст больше чем одна строка, то он может быть продолжен на следующей строке посредством печати символа новой строки с последующей наклонной чертой влево.

<text> может быть опущен. В этом случае все представители идентификатора <identifier> будут удалены из исходного текста программы. Тем не менее, <identifier> рассматривается как определенный и принимает значение 1, если проверяется директивой #if.

Когда задан список параметров <parameter-list>, то он содержит один или более формальных параметров, разделенных запятыми. Каждое имя в списке должно быть уникальным и список должен быть заключен в круглые скобки. Не допускаются пробелы между <identifier> и открывающей скобкой. Имена формальных параметров в тексте <text> отмечают места, куда должны быть подставлены фактические значения. Каждое имя формального параметра может появиться в тексте более одного раза в любом порядке.

Фактические аргументы, следующие непосредственно за идентификатором <identifier> в исходном файле, соответствуют формальным параметрам списка параметров <parameter-list> и модифицируют <text> путем замены каждого формального параметра на соответствующий фактический. Списки фактических и формальных параметров должны содержать одно и то же число аргументов.

Аргументы с побочными эффектами могут стать причиной непредсказуемых результатов. Макроопределение может содержать более одного вхождения данного формального параметра, и если этот формальный параметр представлен выражением с побочным эффектом, то это выражение будет вычисляться более чем один раз.

Примеры:

```
/*.....* example 1 *.....*/
#define WIDTH    80
#define LENGTH   (WIDTH + 10)
/*.....* example 2 *.....*/
#define FILEMESSAGE "Attempt too create file \
failed because of insufficient space"
/*.....* example 3 *.....*/
#define REG1     register
#define REG2     register
#define REG3
/*.....* example 4 *.....*/
#define MAX(x,y)    ((x) > (y)) ? (x) : (y)
/*.....* example 5 *.....*/
```

```
#define MULT (a,b)  ((a) * (b))
```

(1). Определяется идентификатор WIDTH, как целая константа 80, и определяется идентификатор LENGTH, как (WIDTH + 10). Каждое вхождение LENGTH заменяется на (WIDTH + 10), которое в свою очередь заменяется на выражение (80 + 10). Скобки являются важными, поскольку они управляют интерпретацией в операторах, подобных следующему:

```
var = LENGTH * 20;
```

После препроцессирования оператор будет таким: `var = (80 + 10) * 20;`
Значение, которое присваивается, равно 1800. Без скобок значение 80+10*20 равнялось бы 280.

(2). Определяется идентификатор FILEMESSAGE. Определение продолжается на вторую строку путем использования символа "\".

(3). Определены три идентификатора, REG1, REG2, REG3. REG1 и REG2 определены как ключевые слова register. Определение REG3 опущено и, таким образом, любое вхождение REG3 будет удалено из исходного файла. Эти директивы могут быть использованы для того, чтобы обеспечить наиболее важным переменным программы (заданным с REG1 и REG2) задание класса памяти register.

(4). Дано макроопределение, поименованное MAX. Каждое текущее вхождение макро-вызова MAX в исходном файле заменяется выражением ((x)>(y))?(x):(y), в котором формальные параметры x и y заменяются на фактические. Например, вхождение

```
MAX(1,2)
```

заменяется на `((1)>(2)?(1):(2),`
а вхождение `MAX (i, s[i])`
заменяется на `((i)>(s[i]))?(i):(s[i])`

Макро-вызов проще читать, чем соответствующее выражение, которое подставляется. Исходная программа становится проще для понимания.

Замечание: в этом макро аргументы с побочными эффектами могут быть причиной непредсказуемых результатов. Например, макро-вызов MAX (i,s[i++]) заменится на

```
((i)>(s[i++]))?(i):(s[i++])
```

Выражение s[i++] вычисляется дважды. Результат тернарного выражения неопределен, т.к. его операторы могут быть вычислены в любом порядке, а значение переменной i зависит от порядка вычисления.

(5). Определяется макро с именем MULT. Макровызов MULT (3,5) в тексте программы заменяется на (3)*(5). Круглые скобки, в которые заключаются фактические параметры, важны, поскольку они управляют интерпретацией составных аргументов. Например, макровызов MULT (3+4,5+6) заменится на (3+4)*(5+6), что эквивалентно 76. Без скобок результат подстановки 3+4*5+6 равен 29.

Директива #undef , синтаксис:

#undef <identifier>

Директива #undef отменяет текущее определение #define идентификатора <identifier>. Чтобы отменить макроопределение посредством директивы #undef, достаточно задать его идентификатор. Задание списка параметров не требуется.

Директива #undef может быть применена к идентификатору, который ранее не определен. Это дополнительная гарантия того, что идентификатор не определен. Директива #undef обычно используется с директивой #define, чтобы создать область исходной программы, в которой идентификатор имеет специальный смысл. Директива #undef используется также с директивой #if для управления сравнениями участков исходной программы. Пример:

```
#define WIDTH        80
#define ADD(X,Y)     (X)+(Y)
.
.
.
#undef WIDTH
#undef ADD
```

В этом примере директива #undef отменяет определение поименованной константы и макроса. Замечание: в директивах задаются только идентификатор и имя макроса.

9.1.2. #include файлы

Синтаксис:

#include "<pathname>"
#include <<pathname>> стандартные каталоги

Директива #include добавляет содержимое заданного include файла к другому файлу. Определения констант и макросов могут быть организованы в "include" файлах и добавлены к любому исходному файлу #include директивой. "include" файлы также полезны для объявлений общих внешних переменных и составных типов данных. Типы, которые требуется объявить и поименовать однажды, также создаются в #include-файлах.

Директива #include сообщает препроцессору об обработке файла, как если бы этот файл появился в исходной программе в точке, где записана директива. Обработанный текст также может содержать директивы препроцессора. Препроцессор выполняет директивы из нового текста, а затем продолжает процессирование первоначального текста исходного файла.

Имя файла <pathname> - это имя файла с предшествующей спецификацией директория. Синтаксис

спецификации файла зависит от специфики операционной системы, в которой компилируется программа.

Препроцессор останавливает поиск при первом появлении файла с заданным именем. Если задано полное однозначное <pathname>, заключенное в двойные кавычки (" ") или в угловые скобки (< >), то препроцессор ищет только это <pathname> и игнорирует стандартные директории. Если спецификация файла не задана полным <pathname>, но неполная спецификация файла заключена в двойные кавычки, то препроцессор начинает поиск включаемого файла в текущем рабочем директории. Затем препроцессор продолжает поиск в директориях, специфицированных в команданой строке компиляции, и, наконец, ищет в стандартных директориях.

Если спецификация файла заключена в угловые скобки, то препроцессор не будет осуществлять поиск в текущем рабочем директории. Он начнет поиск в директориях, специфицированных в командной строке компиляции, а затем в стандартных директориях.

Директива #include может быть вложенной, другими словами, директива может появиться в файле, поименованном другой #include директивой. Когда препроцессор встречает вложенную #include-директиву, то он обрабатывает файл этой директивы и вставляет его в текущий файл. Препроцессор использует те же самые описанные выше процедуры для поиска вложенных #include-файлов. Новый файл также может содержать директивы #include. Допускается вложение до десяти уровней. Как только вложенные #include-файлы обработаны, препроцессор вставляет этот файл в исходный текстовый файл программы. Примеры:

```
#include <stdio.h> /* example 1 */
#include "defs.h" /* example 2 */
```

(1). В исходную программму вставляется файл, поименованный stdio.h. Угловые скобки сообщают препроцессору, что поиск файла нужно осуществлять в стандартных директориях после поиска в директории, специфицированном в командной строке.

(2). В исходную программу вставляется файл, поименованный defs.h. Двойные кавычки означают, что при поиске файла вначале должен быть просмотрен текущий директорий.

9.1.3. Условная компиляция

В разделе используется синтаксис и использование директив, которые управляют условной компиляцией. Эти директивы позволяют отменить компиляцию частей исходного файла посредством проверки константных выражений или идентификаторов, при которой определяется, нужно ли передавать на выход или пропустить данную часть исходного файла на стадии препроцессирования.

Директивы #if, #elif, #else, #endif

Синтаксис:

```
#if <restricted-constant-expression>
  [<text>]
[#elif <restricted-constant-expression><text>]
[#elif <restricted-constant-expression><text>]
.
.
.
[#else <text>]
#endif
```

Директива #if вместе с директивами #elif, #else и #endiif управляет компиляцией частей исходного файла. Каждой директиве #if в исходном файле должна соответствовать закрывающая директива #endif. Между директивами #if и #endif допускается нуль или более директив #elif и не более одной директивы #else. Директива #else, если она есть, должна быть расположена непосредственно перед директивой #endif.

Препроцессор выбирает один из участков текста-<text> для дальнейшей обработки. Участок <text> - это любая последовательность текста. Он может занимать более одной строки. Обычно это участок программного текста, который имеет смысл для компилятора или препроцессора. Однако, это не обязательное требование. Препроцессор можно использовать для обработки любого текста.

Выбранный текст обрабатывается препроцессором и посылается на компиляцию. Если <text> содержит директивы препроцессора, то эти директивы выполняются.

Любой участок текста, не выбранный препроцессором, игнорируется на стадии препроцессирования и впоследствии не компилируется.

Препроцессор выбирает отдельный участок текста на основе вычисления ограниченного константного выражения <restricted-constant-expression>, следующего за каждой #if или #elif директивой, пока не будет найдено выражение со значением истина (не нуль). Выбирается <text>, следующий за истинным константным выражением, до ближайшего знака номера (#).

Если ограниченное константное выражение не истинно или отсутствует директива #elif, то препроцессор выбирает <text> после записи #else. Если запись #else опущена, а выражение директивы #if ложно, то текст не выбирается.

Ограниченное константное выражение <restricted-constant-expression> не может содержать sizeof выражений, кастовых выражений, перечислимых констант, но может содержать специальные константные выражения defined (<identifier>). Это константное выражение истинно, если заданный идентификатор <identifier> в текущий момент определен, в противном случае выражение ложно. Идентификатор <identifier>, определенный как пустой текст, рассматривается как определенный.

Директивы #if, #elif, #else, #endif могут быть вложенными. Каждая из вложенных директив #else, #elif, #endif принадлежит к ближайшей предшествующей директиве #if. Примеры:

```
/*.....* example 1 *.....*/
#if defined (CREDIT)
   credit ();
#elif defined (DEBIT)
   debit ();
#else
   printerror ();
#endif
/*.....* example 2 *.....*/
#if DLEVEL>5
   #define SIGNAL  1
   #if STACKUSE == 1
            #define STACK  200
   #else
            #define STACK  100
   #endif
#else
   #define SIGNAL  0
   #if STACKUSE == 1
            #define STACK  100
   #else
            #define STACK  50
   #endif
#endif
/*.....* example 3 *.....*/
#if DLEVEL == 0
   #define STACK 0
#elif DLEVEL == 1
   #define STACK 100
#elif DLEVEL > 5
   display ( debugptr );
#else
   #define STACK 200
#endif
/*.....* example 4 *.....*/
#define REG1  register
#define REG2  register
#if defined(M_86)
   #define REG3
   #define REG4
   #define REG5
#else
   #define REG3   register
   #if defined(M_68000)
        #define REG4   register
        #define REG5   register
   #endif
#endif
```

(1). Директивы #if, #endif управляют компиляцией одним из трех вызовов функций. Вызов функции credit компилируется, если идентификатор CREDIT определен. Если определен идентификатор DEBIT, то компилируется функциональный вызов debit. Если ни один идентификатор не определен, то компилируется вызов printerror. Заметим, что CREDIT и credit - это различные идентификаторы в Си.

В следующих двух примерах предполагается, что константа DLEVEL предварительно определена.

(2). Показаны две последовательности вложенных #if, #else, #endif директив. Первая последовательность директив обрабатывается, если DLEVEL >5. В противном случае обрабатывается вторая последовательность.

(3). Используются директивы #elif, #else, чтобы сделать один из четырех выборов, основанных на значении константы DLEVEL. Здесь определяется константа STACK равной 0, 100 или 200, в зависимости от значения DLEVEL. Если DLEVEL>5, то компилируется вызов функции display (debugptr), а константа STACK не определяется.

(4). Директивы препроцессора используются для управления объявлениями спецификатора регистровой памяти register в переносимом исходном файле. Если программа содержит больше объявлений переменных класа памяти

register, чем может предоставить машина, то компилятор не объявит лишние переменные как регистровые. REG1 и REG2 определяются как ключевые слова register, чтобы объявить регистровую память для двух наиболее важных переменных в программе. Например, в следующем фрагменте переменные b и c имеют больший приоритет, чем a или d.

```
func (a)
REG3 int a;
{
  REG1 int b;
  REG2 int c;
  REG4 int d;
  .
  .
  .
  }
```

Когда определен идентификатор M_86, препроцессор удаляет идентификаторы REG3 и REG4 из файла путем замены его на пустой текст. Регистровую память в этом случае получат переменные b и c. Когда определен идентификатор M_68000, то все четыре переменные объявляются с классом памяти register.

Когда не определены оба идентификатора, то объявляются с регистровой памятью три переменные a,b и c.

9.1.4. Директивы #ifdef и #ifndef

Синтаксис:

#ifdef <identifier>
#ifndef <identifier>

Директивы #ifdef и #ifndef выполняют те же самые задачи, что и директива #if, использующая defined(<identifier>). Эти директивы могут быть использованы там же, где используется директива #if, и используются исключительно для компактности записи.

Когда препроцессор обрабатывает директиву ifdef, то делается проверка идентификатора <identifier> на истинность (не нуль).

Директива #ifndef является отрицанием директивы #ifdef. Другими словами, если <identifier> не определен (или его определение отменено директивой #undef), то его значение истинно (не нуль). В противном случае значение ложно (нуль).

9.1.5. Управление нумерацией строк

Синтаксис:

#line <constant>["filename"]

Директива #line инструктирует компилятор об изменении внутренней нумерации строк и имени файла на заданный номер строки и имя файла, для того чтобы сослаться на них в случае ошибок, обнаруженных в процессе компиляции. Номер строки обычно соответствует номеру текущей входной строки. Имени файла соответствует имя текущего входного файла. Номер строки увеличивается после обработки каждой строки. В случае изменения номера строки и имени файла, компилятор игнорирует предыдущие их значения и продолжает обработку с новыми значениями.

Директива #line обычно используется для программной генерации сообщений об ошибках со ссылками на номер строки и имя файла.

Значение константы <constant> в директиве #line - это любая целая константа. Имя файла <filename> может быть любой комбинацией символов, заключенной в двойные кавычки ("). Если имя файла опущено, предполагается, что имя файла осталось текущим.

Текущие номер строки и имя файла доступны через предопределенные идентификаторы __LINE__ и __FILE__. Идентификаторы __LINE__ и __FILE__ могут быть использованы при вставке в исходный файл программного текста выдачи сообщений об ошибке.

Переменная __FILE__ содержит строку, представляющую имя фйла, заключенного в двойные кавычки. Таким образом, нет необходимости заключать идентификатор __FILE__ в двойные кавычки, когда он используется. Примеры:

```
/*.....* example 1 *.....*/
 #line 151 "copy.c"
 /*..... example 2 *.....*/
 #define ASSERT(cond)     if(!cond)\
 {printf("assertion error line %d, file(%s)\n", \
 __LINE__, __FILE__);}else;
```

(1). Номер строки устанавливается равным 151 и имя файла изменяется на copy.c.

(2). В макроопределении ASSERT используются предопределенные идентификаторы __LINE__ и __FILE__ для печати сообщения об ошибке, содержащего координаты исходного файла, если заданное "утверждение" ложно. Заметим, что двойные кавычки при задании предопределенных идентификаторов не требуются.

9.1.6. Генерация сообщения об ошибке

Синтаксис:

#error [<text>]

Приказывает препроцессору выдать сообщение, включающее заданную последовательность лексем <text> и компиляция завершится. Пример:

```
#error Ошибка, эта часть программы не написана!
```

9.1.7. Пустая директива

Не вызывает никаких действий. Синтаксис:

```
#
```

9.1.8. Управляющая строка

Указания компилятору - это команды, которые вставляются в текст исходной программы Си и используются для управления действиями компилятора в определенных частях программы, не влияя на программу в целом. Однако набор указаний компилятору и их смысл зависят от реализации. Неопознанное указание игнорируется.

Синтаксис:

#pragma [<последовательность лексем>]

Пример (для ICC8051 подробно в лекции 1):

```
#pragma language = extended
#pragma memory = data
#pragma function = interrupt
```

Для C51, компилятора Keil Elektronik GmbH, таких указаний – 60. Полностью заменяют опции компилятора.

9.1.9. Генерация предупреждения

Есть для ICC8051.Синтаксис:

#message [<"text">]

Приказывает препроцессору выдать сообщение, включающее заданную последовательность лексем <text> и компиляция продолжается. Пример:

```
#message "Варианты задаются Param"
```

9.2. Операторы препроцессора

Есть два специальных оператора для макрорасширений. **Первый** – это оператор **#**, который ставится перед параметром. Он требует, чтобы подставляемый вместо параметра и знака # (перед ним) текст был заключён в двойные кавычки. При этом в аргументе в строковых литералах и литерных константах перед каждой двойной кавычкой «"» (включая и обрамляющие строку), а также перед каждой обратной наклонной чертой «\» ставится «\». Этот оператор делает строки. Пример:

```
#define mac( p1 ) #p1 "-Param"                    /* становится "p1"+"Param" */
char cWord[] = mac( My );
```

создаёт строку cWord[] ="My-Param".

Второй оператор записывается как **##**. Если последовательность лексем в любого вида макроопределении содержит оператор **##**, то сразу после подстановки параметров он вместе с окружающими его пробельными литерами выбрасывается, благодаря чему склеиваются соседние лексемы, образуя тем самым новую лексему. Оператор не может стоять ни в начале, ни в конце замещающей последовательности лексем. Пример:

```
#define cat( p1, p2 ) p1 ## p2
int cat( var, 123 ) = 123;
```

декларирует и инициализирует переменную var123 = 123.

```
#define mac1(p1)     #p1 "Param"                  /* становится "p1"+"Param" */
#define mac2(p1, p2)  p1+p2##add_this             /* присоединение к p2      */
```

ПРИМЕР ПРОГРАММЫ НА ЯЗЫКЕ СИ

В программе порой требуется осуществлять безусловный переход. Оператор **goto** указывает, что выполнение программы необходимо продолжить, начиная с инструкции, перед которой записана метка перехода. Метку можно поставить перед любой инструкцией в блоке той функции, где находится соответствующий ей оператор goto. **Для переходов между функциями** необходим нелокальный переход (используются средства setjmp.h) (В C++ этот механизм развит для обработки исключений):

Пример 1

```
/*   Пример нелокальных переходов. Вложенные комментарии. Условная компиляция.
     Маленькая рекурсивная программа, вычисляющая числа Фибоначчи.
// вложенные комментарии в IAR Systems, а здесь Keil
*/
#define Param 3
// =======================================
#if Param == 0
#error Ошибка №1, этот фрагмент ещё не написан!
/* . . . . . . . . . . . . */
#elif Param == 1
#error Ошибка №2, этот фрагмент всё ещё не написан!
/* . . . . . . . . . . . . */
#elif Param == 2
#error Ошибка №3, этот фрагмент врядли напишу!
/* . . . . . . . . . . . . */
```

```
#elif Param == 3
// ===== Нелокальные переходы.
#define CnstMyFunc 89                          /* 5040 для факториала */
#pragma small                                  // явная модель памяти.
//#pragma compact
//#pragma large
#include <setjmp.h>
#include <stdio.h>
#include <string.h>
jmp_buf vJumper;
int iN;
static char cResult[10];
static int iValCT, iC = 0;
static int NumFib0 = -1, NumFib1 = 1;
#if CnstMyFunc == 5040
static int Fact( int iNum );
#else
static int Fib( int iNum ) reentrant;
#endif
void fSubroutine( void );

void main( void ) {
/***** Подготовка к работе.
   -включить прогрев; -вкл. ВЧ ген.; -подать высокое. */
  iN = 666;
  strcpy( cResult, "Start!" );
  iValCT = setjmp( vJumper );
/***** Точка входа. Работа блока.
   -30-40 мин прогрев на 5 градусов. */
  if ( iValCT != 0 ) { // не первый вход!
#if CnstMyFunc == 5040
    iN = Fact( iValCT );
#else
    iN = Fib( iValCT );
#endif
    if ( iN >= CnstMyFunc ) {
      strcpy( cResult, "Finish!" );
      goto Lexit;
    }
  }
  else iN = -1;
  iN = sprintf( cResult, "N=%d i=%d", iN, iValCT );
  fSubroutine();
/***** Завершение.
  -снять высокое; выкл. ген.; -вентил. охл. 5 мин. */
Lexit:
  while (1) {}
}

#if CnstMyFunc == 5040
int Fact( int iNum ) {
  int i, prod = 1;
  if ( iNum < 0 ) return 0;
  if ( iNum == 0 ) return 1;
  for ( i = 1; i <= iNum; i++ )
    prod *= i;
  return prod;
}
#else
int Fib( int iNum ) reentrant {
  if ( iNum >= 2 ) return Fib( iNum - 1 )+ Fib( iNum - 2 );
  else if ( iNum == 1 ) return NumFib1;
    else if ( iNum == 0 ) return NumFib0;
```

```
  return 0x8000; // ошибка в параметре!
}
#endif

/* В прерывании longjmp(...) */
void fSubroutine( void ) {
/* ... */
  longjmp( vJumper, ++iC );
/* ... */
}
#else
// ===== Последний вариант.
#message "Этот фрагмент до конца не написан!"
/* . . . . . .
  для IAR Systems
   . . . . . . . */
#endif
```

Для вычислений с факториалом в проект включается лишь файл исходного текста на языке Си – **fib2-1.c**. Для вычисления последовательности Фи-Боначчи в проект нужно добавить файл **STARTUPm.A51**, чтобы вести реентерантный стек, который растёт вниз (как общепринято).

Простая сортировка данных.

Пример 2

```
/*    Пример рекурсии – программа сортировки данных.
      Сортировка К.А.Р. Хоора, 1962г. */
*/
int iTestMas[] = { 5, 3, 1, 4, 2, 0 };
void SortHoor( int iMassiv[], int left, int right );
void main( void ) {
  SortHoor( iTestMas, 1, 4 );
}

void SortHoor( int iMassiv[], int left, int right ) {
  int i, last;
  void SimplSwap( int iMassiv[], int i, int j );
  if ( left >= right ) return;
  SimplSwap( iMassiv, left, ( left + right )/ 2 );
  last = left;
  for ( i = left + 1; i <= right; i++ )
    if ( iMassiv[ i ]< iMassiv[ left ]) SimplSwap( iMassiv, ++last, i );
  SimplSwap( iMassiv, left, last );
  SortHoor( iMassiv, left, last - 1 );
  SortHoor( iMassiv, last + 1, right );
}

void SimplSwap( int iMassiv[], int i, int j ) {
  int iWorkTemp;
  iWorkTemp = iMassiv[ i ];
  iMassiv[ i ]= iMassiv[ j ];
  iMassiv[ j ]= iWorkTemp;
}
```

Для заданного массива выбирается один элемент, который разбивает все остальные на два подмножества – те, что меньше его, и те, что не меньше его. Эта же процедура применяется и к двум полученным подмножествам.

Сортировка Шелла (1959 г.)

Основная идея – на ранних стадиях сравниваются далеко отстоящие друг от друга, а не соседние элементы, как в обычных перестановочных сортировках. Это приводит к быстрому устранению массовой неупорядоченности, благодаря чему на более поздней стадии остаётся меньше перестановок. Интервал между сравниваемыми элементами постепенно уменьшается до единицы, и в этот момент сортировка сводится к обычным перестановкам соседних элементов.

Пример 3

```
    /* Сортировка Шелла. */
    int iTestMas[] = { 5, 3, 1, 4, 2, 0, -1, -3 };
    static void ShellSort( int iMassiv[], int iSizeM );
```

```
void main( void ) {
  ShellSort( iTestMas + 1, 7 );
}
void ShellSort( int iMassiv[], int iSizeM ) {
  int gap, i, j, iWorkTemp;
  for ( gap = iSizeM >> 1; gap > 0; gap >>= 1 )
    for ( i = gap; i < iSizeM; i++ )
      for ( j = i - gap; j >= 0 && iMassiv[ j ]> iMassiv[ j + gap ]; j -= gap ) {
        iWorkTemp = iMassiv[ j ];
        iMassiv[ j ]= iMassiv[ j + gap ];
        iMassiv[ j + gap ]= iWorkTemp;
      }
}
```

Вычисление индекса (j + gap) повторяется 3 раза в цикле. Для ускорения индекс лучше считать один раз.

```
void ShellSort( int iMassiv[], int iSizeM ) {
  int gap, i, j, jGap, iWorkTemp;
  for ( gap = iSizeM >> 1; gap > 0; gap >>= 1 )
    for ( i = gap; i < iSizeM; i++ )
      for ( j = i - gap; j >= 0 && iMassiv[ j ]> iMassiv[ jGap = j + gap ]; j -= gap ) {
        iWorkTemp = iMassiv[ j ];
        iMassiv[ j ]= iMassiv[ jGap ];
        iMassiv[ jGap ]= iWorkTemp;
      }
}
```

В перестановке также несколько раз вычисляются элементы массивов (их адреса). Устранить это на ассемблере.

ПЕРЕМЕННЫЕ ДИНАМИЧЕСКОЙ ПРОДОЛЖИТЕЛЬНОСТИ.

Для ведения динамических переменных требуется включить в исходный текст файл заголовков **stdlib.h** и воспользоваться традиционными функциями ведения динамических переменных:

```
void  *malloc (unsigned int size);                    // выделить память в байтах
void  free (void  *p);                                // освободить память
void  *realloc (void  *p, unsigned int size);         // перераспределить память
void  *calloc (unsigned int size, unsigned int len);  // выделить память в переменных
```

Чтобы работал менеджер динамической памяти, его требуется инициализировать перед использованием указанных выше функций. Менеджер ведёт односвязный список свободных блоков памяти в выделенном регионе. Задать исходный регион памяти нужно с помощью функции:

```
int  init_mempool (void  *p, unsigned int size);
```

Пример использования динамических переменных с выделением необходимого региона в памяти XDATA:

```
#pragma CODE
//#pragma SRC
//===== динамические переменные
#include <stdlib.h>
#include <absacc.h>
int g_iMas[ 10 ];
int * g_pi1, * g_pi2, * g_pi3;
//int xdata g_iXV;
int xdata g_iXMas[ 50 ];

//***** main *****
void main ( void ) {
  int * pi1, * pi2, * pi3;
//  init_mempool( &XBYTE[ 0x10 ], 0x100 );
  init_mempool( g_iXMas, sizeof( g_iXMas ));
  g_pi1 = malloc( 10 );
  g_pi2 = malloc( 20 );
  g_pi3 = malloc( 30 );
  *g_pi1 = 10;
  *g_pi2 = 20;
  *g_pi3 = 30;
//
  pi1 = malloc( 11 );
  pi2 = malloc( 21 );
  pi3 = malloc( 31 );
//
  free( g_pi1 );
  free( g_pi2 );
  free( g_pi3 );
  while( 1 );
}
```

МОДЕЛЬ ПЕРЕКЛЮЧАЕМЫХ БАНКОВ ПАМЯТИ КОДА И ЕЁ ПРОГРАММИРОВАНИЕ НА ЯЗЫКЕ СИ.

Ограничение размера памяти для больших программ до 64 Кбайт в стандарте MCS-51 преодолимо. Для этого необходимы некоторые аппаратные доработки разрабатываемого контроллера, а именно: необходим источник до-

полнительных адресных разрядов и их дешифратор для управления выборкой кристалла дополнительных микросхем памяти. Для адресации могут использоваться свободные разряды порта P1, например.

План работы.

Компиляторы и линковщики системы программирования от Keil Elektronik GmbH полностью поддерживают модель переключаемых банков. Работа с переключаемыми банками памяти включает 5 следующих шагов.

1 шаг. Задать аппаратное переключение банков и распределить их адресное пространство. Для адресации можно использовать порт P1, регистры, адресуемые на внешнюю память данных, и тому подобное. Пример представлен на рис. 1. Общая область начинается с нулевого адреса, а дополнительные банки занимают старшие адреса за общей областью.

В общую область входят традиционно вектора прерываний и сброса системы, обработчики прерываний, константы общего пользования и строки текста, сегменты общего применения, программа, управляющая переключением банков.

Система поддерживает до 32 банков памяти. Если необходимо больше, то программную поддержку необходимо писать самому пользователю.

Пример памяти с 4 переключаемыми банками

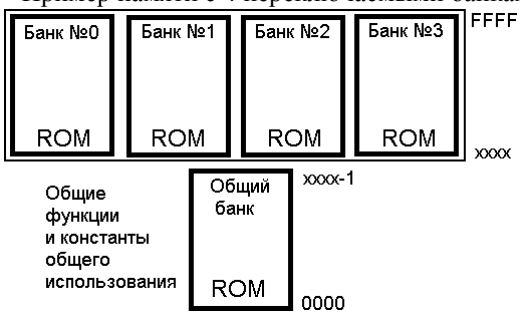

Рис. 1.

2 шаг. Описать банки памяти. Конфигурация их задаётся в диалоге на закладке **Options for Target – Target**. Задаётся число банков и их адресное пространство. Это конфигурация целевой программы. В примере на рис. 1 задано 4 банка памяти. Везде и всегда общий банк – самостоятельная часть кода.

3 шаг. Задать управление переключением банков. В библиотеке Си предоставляется файл управления до 32 банков в трёх различных режимах – **L51_BANK.A51** . Включить этот файл в проект и желательно в отдельную группу. Сконфигурировать этот файл под свою задачу.

4 шаг. Определить содержимое каждого банка и задать свойства всех банков. Содержимое банков лучше задавать в проекте соответственно в раздельных группах для каждого банка. Исходные файлы или вся группа для банка специфицируется в диалоге **Options - Properties** . Вместилище генерируемого кода группы или отдельного исходного файла задаётся **Common** или **Bank #x** соответственно.

5 шаг. Если необходимо, распределить по банкам отдельные сегменты констант. Линковщик по умолчанию распределяет сегменты констант в общую область. Если эти константы применимы только в одном банке, то можно уменьшить размер общей области и поместить сегмент в банке, где он только и используется. Для этого используется директива линковщика **BANKx** в диалоге на закладке **Options for Target – L51 Misc**. Например, в поле Misc controls набраны две директивы:

```
BANK0 ( CONSEG1 (0x8000), CONSEG2 )
BANK1 ( CONSEG3 (0x8000) )
```

которые распределяют три сегмента констант в двух банках: два в нулевом банке и один в первом.

В заключение, если разрешить выбор "**Create HEX File**" на закладке диалога **Option for Target – Output** , то линковщик сгенерирует 16-ый код для программатора для каждого банка отдельно.

Переключение банков.

Основную нагрузку работы по поддержанию модели переключаемых банков выполняет линковщик. Линковщик генерирует специальный код для вызовов процедур из разных банков памяти кода. Вызовы в пределах одного банка остаются в прежнем виде, как их выполнил компилятор.

Межбанковские вызовы собираются в отдельную таблицу INTRABANK CALL TABLE, размещённую в сегменте **?BANK?SELECT**. В таблице одна команда вызова функции заменяется на две. Так вызов функции MyFunc

```
                    CALL   MyFunc
```

заменяется на

```
                    CALL   ?MyFunc?1
```

где новый вход в таблице вызовов имеет следующий вид:

```
?MyFunc?1:    MOV    DPTR, # MyFunc
              JMP    ?B_BANKn
```

В переключении участвуют два типа функций: **?B_BANKn** и **?B_SWITCHn**, которые программист обеспечивает линковщику для соответствующих переключений. Функция **?B_BANKn** запоминает в стеке адрес функции переключателя вызывающего блока. Затем заносит в стек регистр DPTR и вызывает функцию переключатель для вызываемого блока. Функция переключатель **?B_SWITCHn** выполняет аппаратное переключение к блоку **"n"** и вызывает функцию по адресу из макушки стека.

Функции переключения определяются в файле **L51_BANK.A51** из каталога \LIB в нужном количестве.

Конфигурирование модели переключаемых банков.

Для конфигурации модели в файле **L51_BANK.A51** необходимо задать основные параметры. Файл содержит множество макросов, которые генерируют необходимые функции переключения. Редактируется начало файла, фрагмент которого приводится ниже (обязательные 4 параметра):

```
$NOMOD51 NOLINES
$NOCOND
;------------------------------------------------------------------------------
;  This file is part of the BL51 / LX51 Banked Linker/Locater package
;  Copyright (c) 1988 - 2000 Keil Elektronik GmbH and Keil Software, Inc.
```

```
;  Version 2.06 (Code and Variable Banking for Class 8051 Derivatives)
;------------------------------------------------------------------------------
;*********************** Configuration Section ******************************
?B_NBANKS        EQU  4     ; Определяет максимальное число банков в модели     *
;                           ; The following values are allowed: 2, 4, 8, 16, 32 *
;                           ; the max. value for ?B_BANKS is 32                 *
;                             Режим переключения банков                        *
?B_MODE          EQU  4     ; 0 for Bank-Switching via 8051 Port                *
;                           ; 1 for Bank-Switching via XDATA Port               *
;                           ; 4 for user-provided bank switch code              *
;                                                                               *
?B_RTX           EQU  0     ; 0 for applications without RTX-51 FULL            *
;                           ; 1 for applications using RTX-51 FULL              *
;                                                                               *
?B_VAR_BANKING   EQU  0     ; Enable Variable Banking in XDATA and CODE memory  *
;                           ; 0 Variable Banking is disabled                    *
;                           ; 1 XDATA and CODE banking with same address lines  *
;                           ; 2 XDATA uses a different banking port             *
;                                                                               *
IF  ?B_VAR_BANKING <> 0;                                                        *
;-------------------------------------------------------------------------------*
; if ?B_VAR_BANKING is enabled define the following values                      *
;                                                                               *
?B_COMMON_XRAM   EQU  0     ; Specify if your hardware has uses XDATA RAM       *
;                           ; 0 no common RAM: complete 64KB XDATA is banked    *
;                           ; 1 all XDATA variables are located to common RAM   *
;                                                                               *
?B_INTR_ACCESS   EQU  0     ; 0 unlimited access to XDATA memory in interrupts  *
;                           ; 1 interupts access to XDATA variables is allowed  *
;                           ;   after call to  ToDo                            *
;                                                                               *
;-------------------------------------------------------------------------------*
ENDIF;                                                                          *
; ...........
```

?B_NBANKS задаёт общее число банков (по степеням 2) в программируемой модели. Обязательно!
?B_MODE задаёт режим переключения банков. Обязательно!
?B_RTX указывает на использование операционной системы реального времени.
?B_VAR_BANKING указывает на банки для переменных.

Пример.

Стандартный пример из каталога .\C51\EXAMPLES\Bank_EX1. *Пример 4*

```
/*---------------------------------------------------------------------------  1-ый файл C_BANK0.C  */
#include <stdio.h>
extern void func2(void);

void func0(void) {
 printf("FUNCTION IN BANK 0 CALLS A FUNCTION IN BANK 2 \n");
 func2();
}

/*---------------------------------------------------------------------------  2-ой файл C_BANK1.C  */
#include <stdio.h>
extern void func2(void);

void func1(void) {
 printf("FUNCTION IN BANK 1 CALLS A FUNCTION IN BANK 2 \n");
 func2();
}

/*---------------------------------------------------------------------------  3-ий файл C_BANK2.C  */
#include <stdio.h>

void func2(void) {
 printf("THIS IS A FUNCTION IN BANK 2! \n");
```

```
}

/*-------------------------------------------------------------------------     4-ый файл C_ROOT.C     */
#include <stdio.h>
#include <reg51.h>
extern void func0(void);
extern void func1(void);

void main(void) {
/* INITIALIZE SERIAL INTERFACE TO 2400 BAUD @12MHz */
  SCON = 0x52;          /* SCON */
  TMOD = 0x20;          /* TMOD */
  TCON = 0x69;          /* TCON */
  TH1 =  0xf3;          /* TH1 */

  printf("MAIN PROGRAM CALLS A FUNCTION IN BANK 0 \n");
  func0();
  printf("MAIN PROGRAM CALLS A FUNCTION IN BANK 1 \n");
  func1();

  while(1);
}

/*------------------------     5-ый файл управляет переключением банков, это – L51_BANK.A51 .
```

ЛАБОРАТОРНАЯ РАБОТА 2. «ИЗУЧЕНИЕ ИНСТРУМЕНТАЛЬНЫХ СРЕДСТВ ПРОГРАММИРОВАНИЯ MCS-51 НА ЯЗЫКЕ СИ»

Часть 1.

1). Короткие программы – изучить затраты в размере кода и в скорости выполнения программы при доступе к различным областям памяти: DATA, IDATA, XDATA, PDATA, CODE (для переменных и через указатели на них, статической и динамической продолжительности). Заполнить табл. 1.

2). Короткие программы – изучить затраты в размере кода и в скорости выполнения программы для комплексного умножения чисел различных типов от char до long double разными алгоритмами. «Без оптимизации и с оптимизацией» по коду или скорости. Рассмотреть случай – число iB константа. Примеры алгоритмов:

iC0 = iA0 *(iB0 + iB1);	iC0 = iA0 *(iB0 - iB1);	iC0 = iB0 *(iA0 + iA1);	Обычный.
iC1 = iA1 *(iB0 - iB1);	iC1 = iA1 *(iB0 + iB1);	iC1 = iA0 *(iB0 - iB1);	
W2 = iB1 *(iA0 + iA1);	W2 = iB1 *(iA0 - iA1);	W2 = iA1 *(iB0 + iB1);	
iC0 -= W2;	iC0 += W2;	iC1 = iC0 - iC1;	
iC1 += W2;	iC1 += W2;	iC0 -= W2;	

3). Выполнить программу для изучения нелокальных переходов. Затем разбить её на два файла, раздельно компилировать и скомпоновать. Исследовать запросы стека для рекурсивных вызовов (сколько значений последовательности Фибоначчи посчитает программа. Исследовать разные классы памяти.).

4). Изучить затраты в размере кода и в скорости выполнения программы для случая стека в расширенной памяти для вызова функций по сравнению с обычными вызовами.

Часть 2.

1). Реализовать программу сортировки методом Хоора. Исследовать разные классы памяти.

2). Реализовать программу сортировки методом пузырька. Сравнить эффективность методов.

3). Реализовать сортировку Шелла.

4). Исследовать затраты памяти для ведения динамических переменных. Сколько можно создать переменных различных типов в регионе лекционного примера.

5). Все программы собрать в много банковом приложении, где каждый банк отвечает отдельную функциональность.