Программирование на Си для PIC

Я ни раз задавал сам себе вопрос, с какого бы языка начинать изучение. Твёрдо отвечаю – Си, т.к. в ассме много рутины и условностей, что лишает творчества. Постоянно надо проверять и перепроверять себя, а не забыл ли ты то или иное действие. В ассме есть свои неоспоримые преимущества, но о них потом, т.к. это почувствовать можно только на практике. С языком определились. Реально Си учить не надо. Я вам так скажу – мой Си это условно десять пазлов и море логики, которые я комбинирую. Можно ничего не знать, важно понимать механизм, т.е. что на что влияет и к чему приводит. Это как игра в тетрис в котором нужно лишь крутить фигуры и плотнее их ставить. Если вы играли в тетрис (не уверен что вы знаете эту игру), то вы легко поймете что такое Си.

Далее о макете (макетной плате). И на эту тему мне задавали вопрос. И пришел к выводу, что человеку, который не первый день в электронике делать какую-то плату или платку с кнопками и светодиодами не интересно. Школьнику мигалка, пищалка и кнопка будут интересны. Но не взрослому человеку. Тем более всё это можно сделать в Протеусе. Протеус изучается за 30 мин. Тогда вы меня спрашиваете, а что же сделать? Сделать практическое устройство по которому у вас будет цель – цель доделать это устройство до конца. Это самый главный психологический стимул.

Из каких компонентов должно быть устройство? Несколько кнопок (хоть десяток), семисегментные индикаторы 2-5 разрядов, микроконтроллер PIC16F628A (или без А) (на этом микроконтроллере можно много фантазировать), ну и оставить 1-2-3 свободные линии, чтобы что-то внешнее подключить или управлять. У меня всё начинается с идеи и вопроса что собрать и подключить к МК, и сразу думаю, а как это будет подключаться к МК и может ли работать такое подключение. Ну и собственно процесс рисования печатной платы идет в параллели. Необходимо знать и учитывать при рисовании, что не все ножки одинаково работают. И именно это важное начальное условие расписывается в самоучителе с самого начала.

В связи с этим ваша задача сейчас по моему самоучителю разобраться с выводами (ножками), как, какие, в какую сторону, при каких условиях работают эти вывода. При всей простоте задачи вы столкнетесь с массой других несложных вещей, которые нужно изучить.

Что в итоге вы получите?
1) Понимание как, что и с чем соединять.
2) Как управляются эти соединения на элементарном уровне.
3) Начнете привыкать к интерфейсу среды разработки.
4) Начнете изучать структуру текста программы.
5) Начнете понимать механизм работы программы.

Важное условие – вдумчиво читать подряд. Вдумчиво и подряд. Возможно перечитывать.
Мир вам.

Введение. Взаимосвязь Си и Ассемблера. Как учить Си?

Как работает МК. Программа в МК. Области памяти в МК.

Байты и биты.

МК и текст программы. Типы данных. Переменные и константы.

Как и какие микроконтроллеры будем программировать?

Установка MPLAB 8.30 и интеграция PICC Compiler 9.50.

Создание проекта с помощью волшебника (wizard)

Программа №1. «Рыба».

Физиололгия работы программы в МК.

Как работают функции. Комментарии.

Программа №2. «Продолжение рыбы». Именование портов и линий

Регистры портов. Определение направлений работы линий.

Ифы, форы, вайлы или основы интеллекта. Истина не ложь

if-else (если-иначе)

for (в течение)

while (пока)

do-while (делать пока)

switch-case-break (выбрать набор и выйти)

return (возврат)

Избыточный займ и переполнение

Составление проекта из нескольких файлов исходников

Массивы.

Динамическая индикация. Прерывания. Структуры

Введение. Взаимосвязь Си и Ассемблера. Как учить Си?

Я не заканчивал технических вузов, у меня не было преподавателей по программированию* и электронике, но у меня было дикое желание научиться чему-то большему, чем поменять лампочку в китайском фонарике. Данный материал для тех, кто хочет думать по-новому, понимать суть сложных вещей и осознано осваивать микроконтроллеры и язык Си. С нуля. Я не знаю с каким уровнем подготовленности вы читаете эти строки, и поэтому Си и микроконтроллеры я буду здесь освещать именно с нуля.

(*Здесь я слукавил; у меня есть учитель - Юрий Анатольевич Петрик из г.Винницы; доброго здоровья ему и процветания).

Взаимосвязь Си и Ассемблера.

Си и Ассемблер – языки программирования. И много бестолковых споров что лучше. Каждый язык прекрасен по-своему, и у каждого свои преимущества. И вы должны принять на веру, что эти языки по сути своей работы одинаковы, т.е. работают в данном контексте с одним и тем же предметом – микроконтроллером (далее МК). Те, кто знает как работает Ассемблер, начнут понимать Си прочитав без заучивания этот самоучитель.

Как учить Си?

Я начинал с ассемблера. Си «начал понимать» через два года после ассемблера. На самом деле, Си я понял за пол часа. Хотя до этого я читал много материала по Си и даже классиков «Язык программирования Си (Б.Керниган Д.Ритчи)».

Мне хотелось бы, чтобы вы повторили мой путь и начали с Ассемблера. Зная как работают команды Ассемблера, вы будете более грамотно составлять программу на Си. Если есть время и желание, прочитайте мой самоучитель по Ассемблеру: «PIC микроконтроллеры: быстрый старт с нуля». Ну а если нет времени или возможности быть более продвинутым, читайте этот материал. Но чтобы быть «кулл хацкером» нужно знать «ассм».

Ни в одной книжке я не встретил описания последовательности действий по пониманию программирования на Си. Язык Си не нужно учить. Нужно понять как он работает. И после того как вы поймете, вы начнете задавать вопросы и научитесь

находить ответы. Ну а чтобы понять как учить и как работает Си необходимо понять –

Как работает МК. Программа в МК. Области памяти в МК.

МК в этом тексте также мы будем называть чип, кирпич, кристалл, камень и т.п. МК это микросхема с набором выводов (ножек, пинов, иголок, pin – англ. иголка), где как минимум два – это вывода питания (плюс и минус 5 вольт). Остальные ножки в зависимости от модели МК могут распознавать внешние входящие сигналы, например, подачу напряжения или факт замыкания кнопки относительно линий питания. Также ножки могут устанавливать исходящие сигналы. Почти все ножки могут работать в двух направлениях, и на вход и на выход, т.е. оценивать внешние входящие сигналы либо устанавливать исходящие сигналы.

Исходящие сигналы бывают Высокого или Низкого Логического Уровней (ВЛУ и НЛУ). Иначе говоря, НЛУ это «ноль» - сигнал близкий к нулю вольт, а ВЛУ это «единица» - сигнал близкий к напряжению питания, т.е. 5 вольт. Входящие сигналы обрабатываются, а исходящие формируются с помощью –

Программа в МК.
Программа в МК определяет алгоритм (т.е. последовательность) распознавания входящих сигналов и формирования исходящих сигналов определенной длительности, определенного уровня, определенной последовательности и на определенных ножках. Иначе говорят, что программа дергает ножки. Эти процессы неразрывно связаны с понятием времени и частотой тактирования кристалла. Но это не так важно на этом этапе обучения. Более важен вопрос об –

Области памяти в МК.
Каждый МК, который мы будем рассматривать на практике, имеет три области памяти:
1) **память программ** или флеш-память, область куда записываются строчки текста программы в момент «прошивания» МК;
2) **оперативная память** или регистровая память, область, в которую во время работы МК записываются, хранятся и изменяются байты, пока на МК подается питание. Сброс питания приводит к сбросу оперативной памяти;
3) **энергонезависимая память** ПЗУ или EEPROM, область, в которую в момент прошивания и/или во время работы МК записываются, хранятся и изменяются данные. Сброс питания не влияет на содержимое этой памяти.

Области памяти состоят из т.н. ячеек, в которых хранятся байты.

Байты и биты.

Байт – это восемь бит. Бит – это минимальный элемент информации, который может быть равен нулю или единице. Комбинация из восьми битов составляет байт, т.е. восьмиразрядное бинарное число, например 10001101 . Всего на восьми разрядах из нулей и единиц можно составить 256 комбинаций. См. таблицу ниже. Числа в МК хранятся в ячейках, размерностью один байт. МК оперирует байтами и поэтому называется восьмиразрядный. Закономерен вопрос – а если нам нужно больше чем 256 комбинаций? В таком случае используется несколько байтов. Например, набор единиц и нулей в двух байтах даст уже 256*256=65536 комбинаций. И т.д. Далее мы рассмотрим вопрос о –

Таблица соответствия чисел в разных системах счисления
и символов из таблицы ANSI (Windows-1251)

D	B	H	A	D	B	H	A	D	B	H	A	D	B	H	A
0	0000 0000	00		64	0100 0000	40	@	128	1000 0000	80	Ђ	192	1100 0000	C0	А
1	0000 0001	01		65	0100 0001	41	A	129	1000 0001	81	Ѓ	193	1100 0001	C1	Б
2	0000 0010	02		66	0100 0010	42	B	130	1000 0010	82	‚	194	1100 0010	C2	В
3	0000 0011	03		67	0100 0011	43	C	131	1000 0011	83	ѓ	195	1100 0011	C3	Г
4	0000 0100	04		68	0100 0100	44	D	132	1000 0100	84	„	196	1100 0100	C4	Д
5	0000 0101	05		69	0100 0101	45	E	133	1000 0101	85	…	197	1100 0101	C5	Е
6	0000 0110	06		70	0100 0110	46	F	134	1000 0110	86	†	198	1100 0110	C6	Ж
7	0000 0111	07		71	0100 0111	47	G	135	1000 0111	87	‡	199	1100 0111	C7	З
8	0000 1000	08		72	0100 1000	48	H	136	1000 1000	88	€	200	1100 1000	C8	И
9	0000 1001	09		73	0100 1001	49	I	137	1000 1001	89	‰	201	1100 1001	C9	Й
10	0000 1010	0A		74	0100 1010	4A	J	138	1000 1010	8A	Љ	202	1100 1010	CA	К
11	0000 1011	0B		75	0100 1011	4B	K	139	1000 1011	8B	‹	203	1100 1011	CB	Л
12	0000 1100	0C		76	0100 1100	4C	L	140	1000 1100	8C	Њ	204	1100 1100	CC	М
13	0000 1101	0D		77	0100 1101	4D	M	141	1000 1101	8D	Ќ	205	1100 1101	CD	Н
14	0000 1110	0E		78	0100 1110	4E	N	142	1000 1110	8E	Ћ	206	1100 1110	CE	О
15	0000 1111	0F		79	0100 1111	4F	O	143	1000 1111	8F	Џ	207	1100 1111	CF	П
16	0001 0000	10		80	0101 0000	50	P	144	1001 0000	90	ђ	208	1101 0000	D0	Р
17	0001 0001	11		81	0101 0001	51	Q	145	1001 0001	91	‘	209	1101 0001	D1	С
18	0001 0010	12		82	0101 0010	52	R	146	1001 0010	92	’	210	1101 0010	D2	Т
19	0001 0011	13		83	0101 0011	53	S	147	1001 0011	93	“	211	1101 0011	D3	У
20	0001 0100	14		84	0101 0100	54	T	148	1001 0100	94	”	212	1101 0100	D4	Ф
21	0001 0101	15		85	0101 0101	55	U	149	1001 0101	95	•	213	1101 0101	D5	Х
22	0001 0110	16		86	0101 0110	56	V	150	1001 0110	96	–	214	1101 0110	D6	Ц
23	0001 0111	17		87	0101 0111	57	W	151	1001 0111	97	—	215	1101 0111	D7	Ч
24	0001 1000	18		88	0101 1000	58	X	152	1001 1000	98	˜	216	1101 1000	D8	Ш
25	0001 1001	19		89	0101 1001	59	Y	153	1001 1001	99	™	217	1101 1001	D9	Щ
26	0001 1010	1A		90	0101 1010	5A	Z	154	1001 1010	9A	љ	218	1101 1010	DA	Ъ
27	0001 1011	1B		91	0101 1011	5B	[	155	1001 1011	9B	›	219	1101 1011	DB	Ы
28	0001 1100	1C		92	0101 1100	5C		156	1001 1100	9C	њ	220	1101 1100	DC	Ь
29	0001 1101	1D		93	0101 1101	5D	]	157	1001 1101	9D	ќ	221	1101 1101	DD	Э
30	0001 1110	1E		94	0101 1110	5E	^	158	1001 1110	9E	ћ	222	1101 1110	DE	Ю
31	0001 1111	1F		95	0101 1111	5F	_	159	1001 1111	9F	џ	223	1101 1111	DF	Я
32	0010 0000	20		96	0110 0000	60	`	160	1010 0000	A0		224	1110 0000	E0	а
33	0010 0001	21	!	97	0110 0001	61	a	161	1010 0001	A1	Ў	225	1110 0001	E1	б

34	0010 0010	22	"	98	0110 0010	62	b	162	1010 0010	A2	ў	226	1110 0010	E2	в
35	0010 0011	23	#	99	0110 0011	63	c	163	1010 0011	A3	Ј	227	1110 0011	E3	г
36	0010 0100	24	$	100	0110 0100	64	d	164	1010 0100	A4	¤	228	1110 0100	E4	д
37	0010 0101	25	%	101	0110 0101	65	e	165	1010 0101	A5	Ґ	229	1110 0101	E5	е
38	0010 0110	26	&	102	0110 0110	66	f	166	1010 0110	A6	¦	230	1110 0110	E6	ж
39	0010 0111	27		103	0110 0111	67	g	167	1010 0111	A7	§	231	1110 0111	E7	з
40	0010 1000	28	(	104	0110 1000	68	h	168	1010 1000	A8	Ё	232	1110 1000	E8	и
41	0010 1001	29	)	105	0110 1001	69	i	169	1010 1001	A9	©	233	1110 1001	E9	й
42	0010 1010	2A	*	106	0110 1010	6A	j	170	1010 1010	AA	Є	234	1110 1010	EA	к
43	0010 1011	2B	+	107	0110 1011	6B	k	171	1010 1011	AB	«	235	1110 1011	EB	л
44	0010 1100	2C	,	108	0110 1100	6C	l	172	1010 1100	AC	¬	236	1110 1100	EC	м
45	0010 1101	2D	-	109	0110 1101	6D	m	173	1010 1101	AD		237	1110 1101	ED	н
46	0010 1110	2E	.	110	0110 1110	6E	n	174	1010 1110	AE	®	238	1110 1110	EE	о
47	0010 1111	2F	/	111	0110 1111	6F	o	175	1010 1111	AF	Ї	239	1110 1111	EF	п
48	0011 0000	30	0	112	0111 0000	70	p	176	1011 0000	B0	°	240	1111 0000	F0	р
49	0011 0001	31	1	113	0111 0001	71	q	177	1011 0001	B1	±	241	1111 0001	F1	с
50	0011 0010	32	2	114	0111 0010	72	r	178	1011 0010	B2	І	242	1111 0010	F2	т
51	0011 0011	33	3	115	0111 0011	73	s	179	1011 0011	B3	і	243	1111 0011	F3	у
52	0011 0100	34	4	116	0111 0100	74	t	180	1011 0100	B4	ґ	244	1111 0100	F4	ф
53	0011 0101	35	5	117	0111 0101	75	u	181	1011 0101	B5	µ	245	1111 0101	F5	х
54	0011 0110	36	6	118	0111 0110	76	v	182	1011 0110	B6	¶	246	1111 0110	F6	ц
55	0011 0111	37	7	119	0111 0111	77	w	183	1011 0111	B7	·	247	1111 0111	F7	ч
56	0011 1000	38	8	120	0111 1000	78	x	184	1011 1000	B8	ё	248	1111 1000	F8	ш
57	0011 1001	39	9	121	0111 1001	79	y	185	1011 1001	B9	№	249	1111 1001	F9	щ
58	0011 1010	3A	:	122	0111 1010	7A	z	186	1011 1010	BA	є	250	1111 1010	FA	ъ
59	0011 1011	3B	;	123	0111 1011	7B	{	187	1011 1011	BB	»	251	1111 1011	FB	ы
60	0011 1100	3C	<	124	0111 1100	7C	|	188	1011 1100	BC	ј	252	1111 1100	FC	ь
61	0011 1101	3D	=	125	0111 1101	7D	}	189	1011 1101	BD	S	253	1111 1101	FD	э
62	0011 1110	3E	>	126	0111 1110	7E	~	190	1011 1110	BE	s	254	1111 1110	FE	ю
63	0011 1111	3F	?	127	0111 1111	7F	□	191	1011 1111	BF	ї	255	1111 1111	FF	я

Эта таблица вам пригодится не один раз. Скачайте этот файл с таблицей и распечатайте.

МК и текст программы. Типы данных. Переменные и константы.

Мы знаем, что текст программы записывается во флеш-область во время «прошивания» кристалла. Необходимо понимать, что текст программы не меняется и не изменится, пока мы во флеш не запишем новый текст. И именно в тексте программы мы говорим, что нужно делать контроллеру – установить на ножке исходящий сигнал или принять внешний сигнал. Работа с сигналами это лишь небольшая часть работы МК, но самая важная. Другая часть работы заключается в математическом обсчете принятых данных или математическом обсчете исходящих данных. Вот мы и подошли к ключевому термину – «данные».

Типы данных.
Это ни что иное, как числа. Ну, например, число входящих сигналов за определенное время. Или число, характеризующее количество времени между исходящими сигналами. Данные это числа и комбинации чисел. А какие числа/данные можно на Си обсчитывать? Отвечаем – определенной размерности или определенных типов.

Тип	Размер байт	Диапазон
bit (бит)	1/8	0, 1
char (символ)	1	–128 … 127
unsigned shar (символ без знака)	1	0 … 255
int (целое)	2	–32768 … 32767
unsigned int (целое без знака)	2	0 … 65535
long int (длинное целое)	4	–2147483648 … 2147483647
unsigned long int (длинное целое без знака)	4	0 … 4294967295
float (с плавающей точкой)	4	±1,175e-38… ±3,402e38

Т.е. бит может принимать одно из двух значений 1 или 0. Символ – это байт с соответствующим диапазоном значений. Под «целое» выделяется два байта. «Длинному целому» предоставляется четыре байта.

Переменные и константы.
Мы говорили о числах, которые могут записываться, храниться и изменяться. Числа, которые могут изменяться называют переменными. А что такое константы? Это жестко прописанные числа. А где мы жестко прописываем информацию, т.е. не можем её изменить? Правильно во флеш-памяти, а иначе говоря, в памяти программ. Чтобы проще усвоить понятия о переменных и константах рассмотрим выражение x+y=25, где x и y это переменные, а 25 – это константа. Лучше это так понимать.

Теперь предлагаю перечитать выше написанное и попытаться мысленно нарисовать в голове картину в каких-то элементарных образах. Ну и табличку типов, и особенно названия типов на английском заучить. Хотя бы первые пять строчек. В качестве отдыха сейчас следует напрячь Интернет и собрать дополнительный материал.

Как и какие микроконтроллеры будем программировать?

Речь шла о PIC. Какие конкретно? Те, на которые есть русскоязычная документация:
PIC12F629_675.pdf
PIC16F627_628.pdf
PIC16F873_874_876_877.pdf
PIC18F242_252_442_452.pdf

Всё это сокровище находится здесь. Там же вы найдете и другую официальную документацию по работе МК на русском (!) языке. Как минимум следует скачать PIC16F627_628.pdf , т.к. на PIC16F628Амы будем обучаться (дешевый микроконтроллер среднего семейства с возможностью самотактирования от встроенного осциллятора). На файл PIC16F627_628.pdf , т.е. на этот даташит я буду

по-умолчанию ссылаться в этом самоучителе. Сделайте ярлыки с этих даташитов на рабочий стол вашего ПК.

Не переживайте, если вы не нашли в продаже PIC16F628A. Я вам покажу, как легко и непринужденно можно делать миграцию (переносить) программу с одного камня на другой.

Вопрос «как будем программировать» состоит из двух пунктов. Под программированием понимаются процессы написания программы и прошивания.

Писать текст программы на СИ мы будем в программе MPLAB IDE. Я не знаю, сколько прошло времени от момента написания этих строк, до момента вашего прочтения, но уверен что версия MPLAB IDE описываемая здесь уже устарела. У меня MPLAB IDE v8.30 и именно её мы будем здесь рассматривать. Я знаю, что уже выпущена версия v8.40, но я не стал обновляться, т.к. изменения в новой версии для меня не будут критичны. Я советую вам сначала освоить по самоучителю v8.30, а затем интуитивно разобрать юзерские полезности в более старших версиях. Откуда скачивать? С официального сайтаMicrochip из архива программ. Вы должны понимать, что термин «устаревшая версия MPLAB IDE» не относится к языку программирования и не является препятствием для вашего обучения.

Что касается пункта «прошивания», то для этого необходимо изучить материал с этой страницы сайта.

Потребуется компилятор. Компилятор – это программа, позволяющая компилировать (переводить текст) с языка программирования в машинные коды, т.е. в прошивку. MPLAB содержит компилятор Ассемблера. Строго говоря, язык Ассемблера, это машинные коды представленные в виде словесных команд. Для компиляции с языка Си нам потребуется HI-TECH PICC Compiler 9.50. Т.к. «правильный» компилятор стоит денег, в Интернете есть демо-версия с ограниченным объемом выходного кода (совсем правильное здесь). Необходимо отметить, что в комплекте с MPLAB IDE v8.30 идёт PICC Compiler 9.60PL5. Но он также имеет ограничения Omniscient Code Generation not available in Lite mode.

И последнее. Практическое и синтетическое моделирование наших проектов. Для этого нам потребуется программа Proteus. Сейчас у меня версия 7.7 SP2 и, вероятно, что и эта версия уже устарела. Однако, и это не препятствие. На благодатной почве нашего сайта вы можете найти ссылкидля скачивания дистрибутивы и лекарства от жадности, в т.ч. мою лекцию «Моделирование работы микроконтроллеров в Proteus или как зашить ПИК в Протеусе». Гарантирую, что проекты работающие в Протеусе, будут работать в реальном железе (кроме отдельно оговариваемых случаев).

Установка MPLAB 8.30 и интеграция PICC Compiler 9.50.

На моем ПК стоит ОС Windows XP SP3.
Для инсталляции MPLAB 8.30 закрываем все программы, в т.ч. и броузер из которого вы, вероятно, читаете этот текст (прочтите этот абзац до конца и всё закройте). Запускаем инсталлятор Install_MPLAB_8_30.exe Next → соглашаемся с лицензией + Next → полная установка (complete) + Next→ расположение папки «C:\Program Files\Microchip\» + Next → соглашаемся с очередной лицензией + Next → Next → ожидаем завершения установки. В конце предлагается установить PICC Compiler

9.60PL5 (HCPICP-pro-9.60PL5.exe); мы отказываем → Нет . Жмём Finish . Закрываем появившееся окно. Всё просто. Расположение папок здесь оговорил, чтобы у меня с вами было полное соответствие.

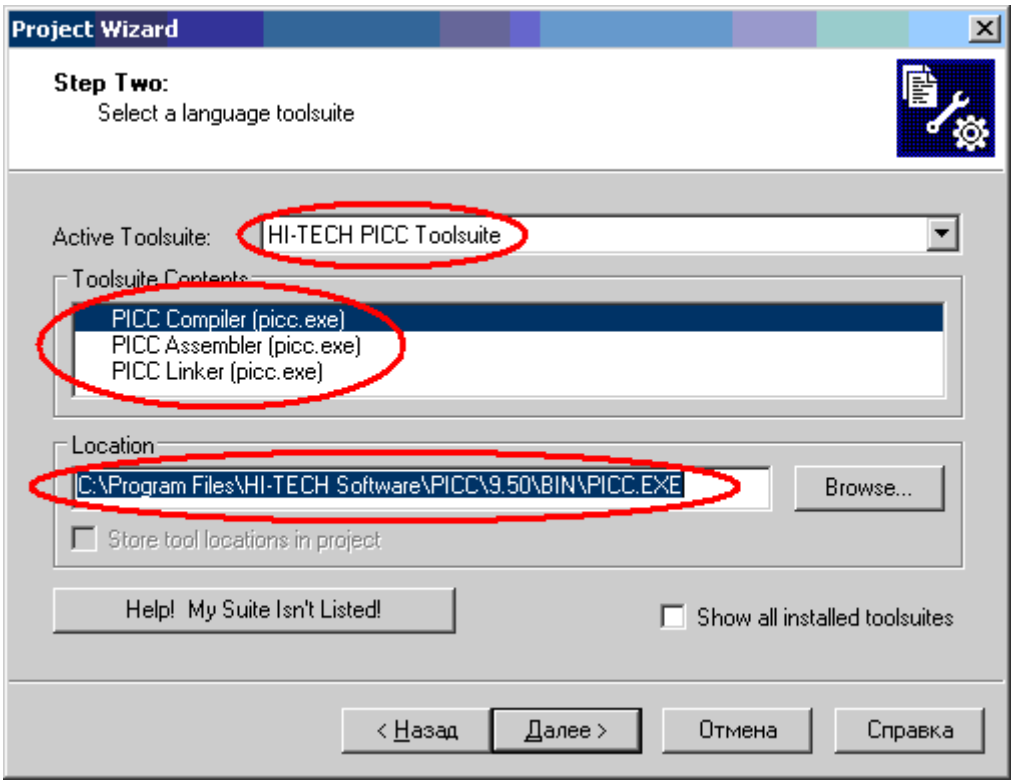
Запускаем HI-TECH_PICC_9.50_PL2.exe. Папка назначения «C:Program Files» + Извлечь. Происходит распаковка. После распаковки запускаем C:\Program Files\HI-TECH Software\PICC\9.50\resources\PICC9.50PL2_REG.reg и добавляем информацию в реестр. Далее запускаем конфигуратор **C:\Program Files\HI-TECH Software\PICC9.50\bin\MPLABConfig.exe** В окне указывается расположение папки **C:\HTSOFT\MPLAB_toolsuites** + Install . В следующем окне стоит галочка и предлагается сразу же запустить MPLAB IDE ; галочку оставляем и жмем Finish. Автоматически запускается оболочка MPLAB IDE.

Смотрю в ваши глаза и вижу тоску. В свое время и мне MPLAB IDE показался тоскливым. Так, создаемпапку «Project» по этому пути C:\Program Files\Microchip\Project. Забудьте про кириллицу (и делательно забыть про пробелы), всё на английском. Кстати – как у вас с английским? Будем учить. По чуть-чуть.

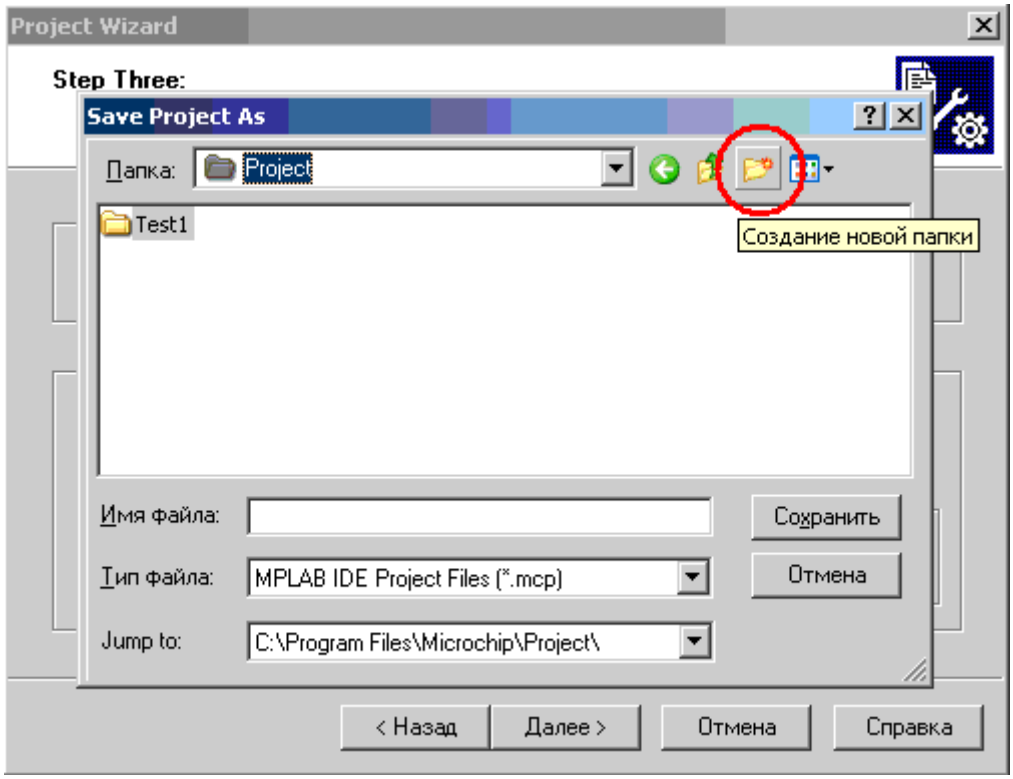
Возвращаемся в окно MPLAB IDE.

Создание проекта с помощью волшебника (wizard)

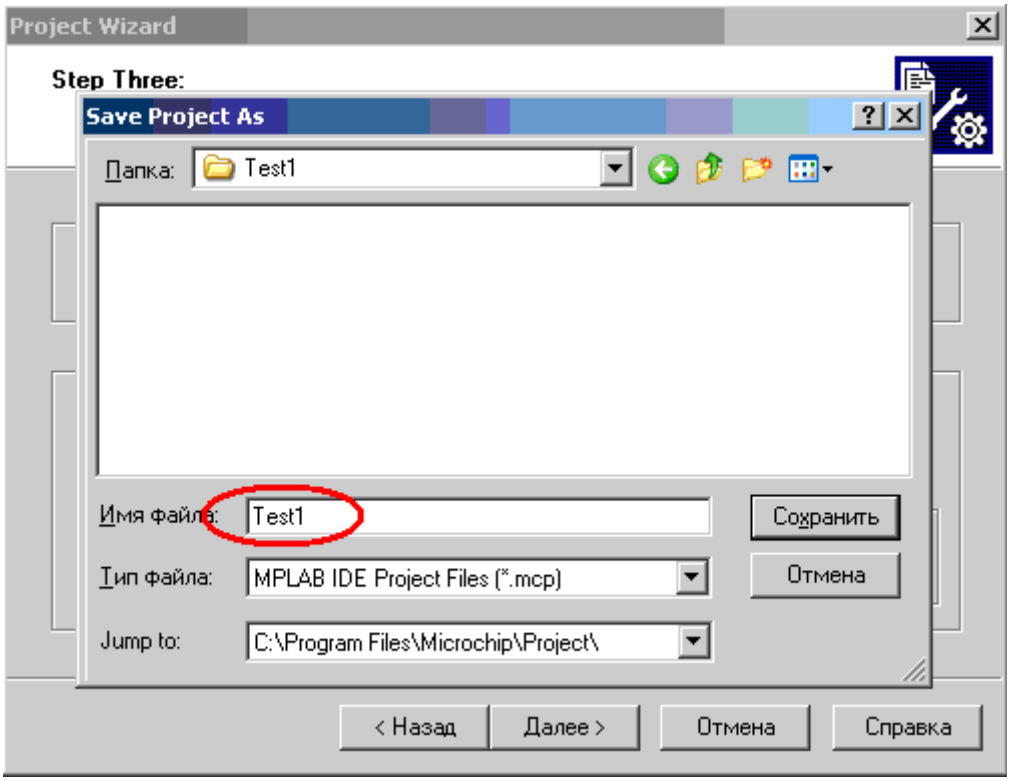
Что такое проект? Это набор файлов, необходимый для организации работ в MPLAB. Не будем себе усложнять жизнь и создадим наш первый и последующие проекты с помощью визарда (дословный перевод – волшебник). Меню Project – Project Wizard… – Далее – выбираем наш кирпич (я выбрал PIC16F628A) + Далее – следующее окно у вас должно быть как на картинке ниже



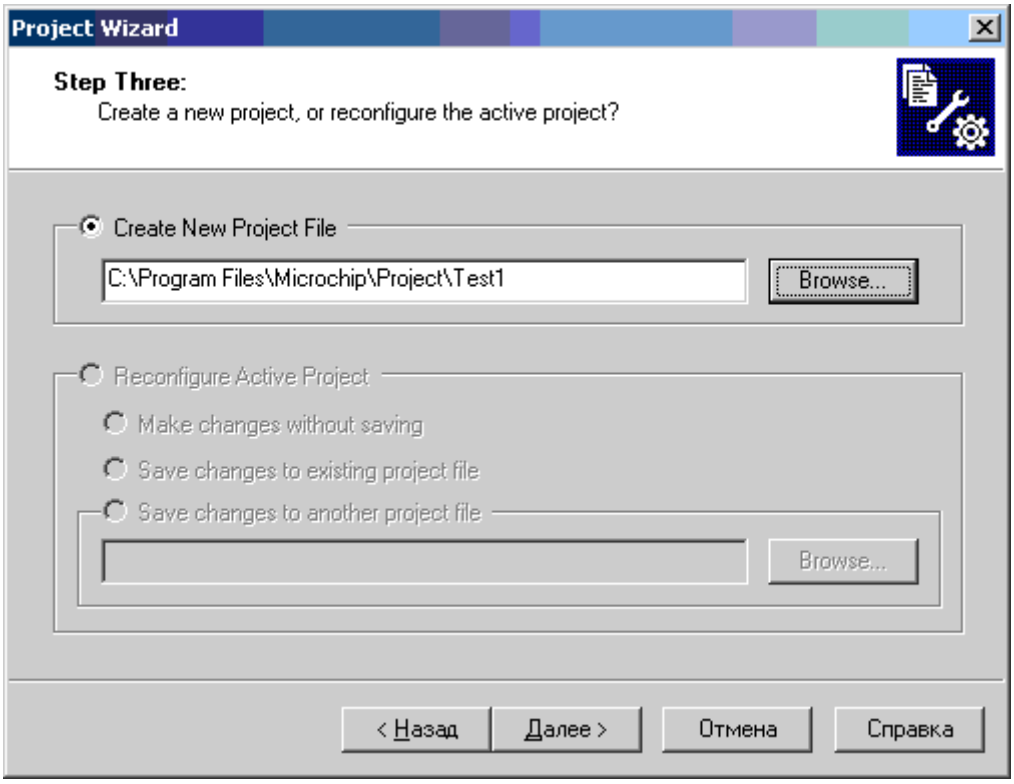
В следующем окне предлагается создать новый проект (мы создадим папку и собственно файл проекта *.mcp). Нажимаем Browse..., в новом окне создаем папку Test1 (по пути C:\Program Files\Microchip\Project\)



Далее заходим в папку Test1 и в поле «Имя файла» пишем имя проекта, которое назовем также Test1.



Нажимаем Сохранить и должно получиться как на картинке ниже.



Нажимаем Далее

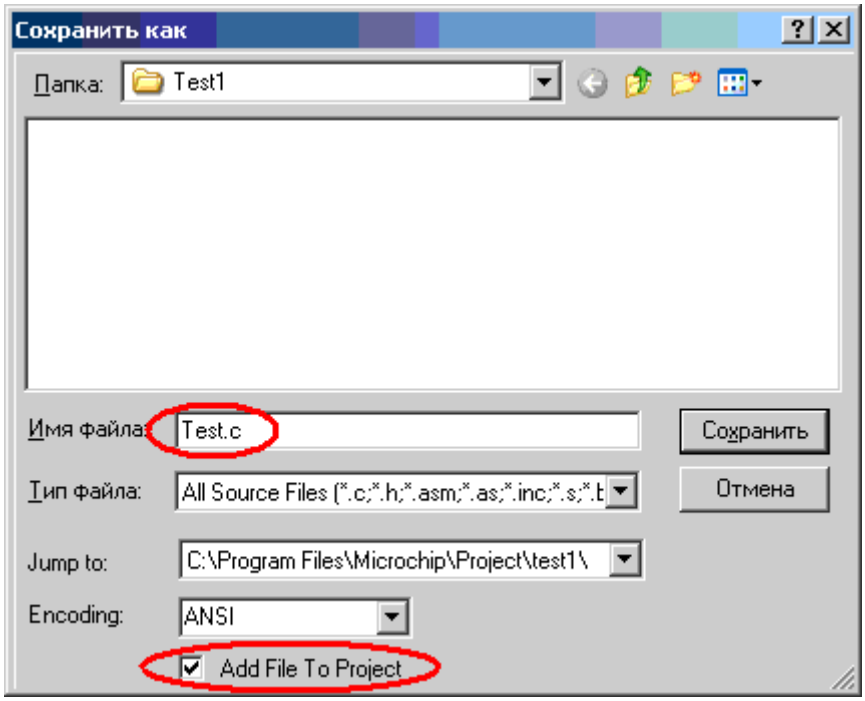
В следующем окне предлагается добавить к проекту файлы (например, исходники или части других проектов), но у нас пока ничего нет и мы нажимаем – Далее – Готово

Запомните! Все имена папок на английском, все имена файлов на английском без пробелов и тире, допускается нижнее подчеркивание. Не рекомендую использовать в именах первым символом цифры (это затем станет строгим правилом при программировании).

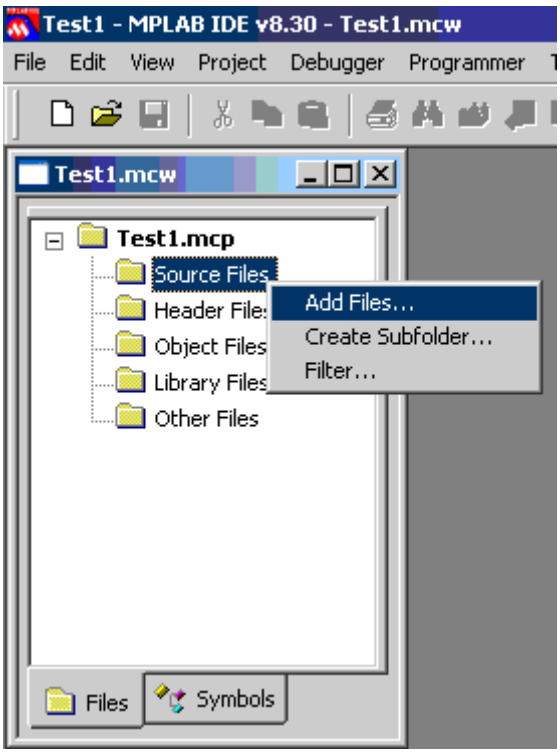
Отлично, мы создали проект с набором файлов. Реально из этого набора файлов самым ценным будет являться файл с расширением *.c (исходник на Си). Проект можно в любой момент сделать с любым именем и вставить текст из исходника. А процесс компиляции текста Си в прошивку (в файл с расширением *.hex) – быстрая процедура в MPLAB, которая выполняется по факту нажатия кнопки F10 клавиатуры.

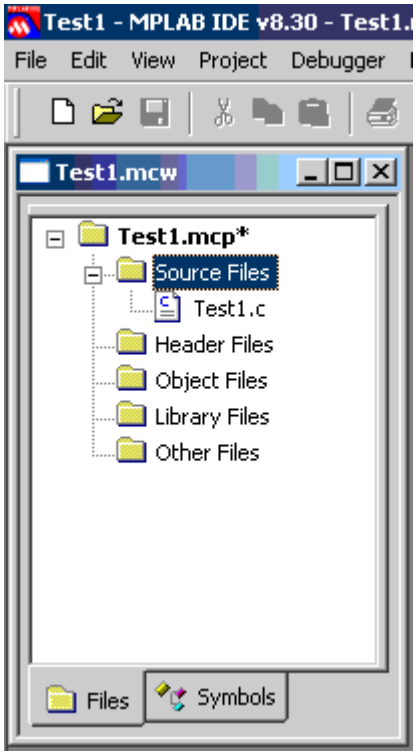
Программа №1. «Рыба».

С чего всё начинается? Мы говорили про исходник с расширением *.c. У нас его нет. Создаем. Нажимаем меню File – New. Появляется окно с именем Untitled. Теперь нам надо сохранить этот файлпод определенным именем в определенном месте и подключить к проекту. Традиционным именем главного Си-файла является имя «main.c». Однако, MPLAB великолепно работает и с другими именами. В предыдущем разделе мы назвали проект как Test1. Файл исходника называем именем проекта (это будет имя «Test1.c»). Нажимаем меню File – Save As… и приходим в путь C:\Program Files\Microchip\Project\Test1 . Далее вводим имя файла **Test1.c** (внимание – обязательно вручную дописываем через точку расширение *.c) и не забываем поставить галочку Add File To Project (Добавить Файл В Проект) + Сохранить.



Представим ситуацию, что вы забыли поставить эту галочку. Не беда.





Правой кнопкой мыши в окне дерева проекта щелкаем на ветке Source File (англ. – источники; сленг – сырцы) и выбираем пункт Add Files…, затем находим и открываем наш файл Test.c. В дереве появляется наш файл. Это признак того, что файл подключен к проекту.

В итоге по пути C:Program FilesMicrochipProjectTest1 у нас сформируется 4 файла с одним и тем же именем, но с разными расширениями (`Test1.mcp , Test1.mcs , Test1.mcw , Test1.c`).
(Пуск – Настройка – Панель управления – Свойства папки – закладка Вид – снять галочку Скрывать расширения для зарегистрированных типов файлов).

Ну а теперь пишем программу «Рыба».

```
#include <pic.h>

__CONFIG (INTIO & UNPROTECT & LVPDIS & BOREN & MCLRDIS & PWRTEN &
WDTDIS);

void main (void)
{
}
```

Этот текст набираем в окне Test.c и затем нажимаем F10. Происходит компиляция и открывается окноOutput. Если внизу в окне Output вы видите фразу типа

```
Loaded C:Program FilesMicrochipProjectTest1Test1.cof.
BUILD SUCCEEDED: Thu Apr 23 10:17:35 2009
```

то вы успешно откомпилировали свою рыбу. Это должно нас радовать. Нами создана первая прошивка (hex-файл), которая располагается в папке с проектом.

Что делать с получившейся рыбой? Разбирать по косточкам.
Строго говоря, для успешной компиляции было бы достаточно текста.

```
main ()
{
}
```

Текст можно записать в одну строчку
`void main (void){}`
или так
`main(){}`

Эти особенности мы разберем чуть позже.

Так, первая строчка `#include <pic.h>` даёт указание компилятору, что всё что есть в файле pic.h«принять к сведению». И этот файл находится по пути `C:\Program Files\HI-TECH Software\PICC9.50\include` . Этот файл иначе называют заголовочным файлом. Давайте откроем`pic.h` из MPLAB. Когда мы создавали проект с помощью визарда, мы выбирали кристалл. В момент нажатия F10 компилятор «смотрит», какой мы выбрали кристалл с помощью визарда, затем по

указанию `#include <pic.h>` зачитывает текст из `pic.h` , в этом тексте он «спотыкается» на записи

```
#if defined(_16F627A)   || defined(_16F628A)    ||
defined(_16F648A)
#include    <pic16f62xa.h>
```

и уходит в файл `pic16f62xa.h`. Файл `pic.h` можно закрыть и затем открываем файл `pic16f62xa.h`. Зачем я всё это рассказываю? Чтобы вы осознано понимали взаимосвязь всех записей.

Следующая строчка – конфигурирование кристалла

```
__CONFIG (INTIO & UNPROTECT & LVPDIS & BOREN & MCLRDIS & PWRTEN &
WDTDIS);
```

Что такое конфигурирование? Это установка определенных режимов работы МК. Более подробно о конфигурировании МК смотрим в документе `PIC16F627_628.pdf` на стр.88. (разместите ярлык этого даташита на рабочем столе).

Что означает фраза `__CONFIG` ? Это так называемая директива (указание) на установку битов конфигурации. А откуда взялись фразы `INTIO & UNPROTECT & LVPDIS & BOREN & MCLRDIS & PWRTEN& WDTDIS` ? Они у нас из файла `pic16f62xa.h`. Этим фразам конфигурирования сопоставлены 16-ричные числа. `0x3FFC 0x3FFF 0x3F7F 0x3FFF` и т.д. С тем же успехом мы могли бы написать через логические «и» (&)

`__CONFIG (0x3FFC & 0x3FFF & 0x3F7F & 0x3FFF &` и т.д.

Согласитесь, удобнее написать осмысленные фразы, чем слабопонятные цифры. Если мы захотим перенести программу на другой кирпич, то там фразам могут соответствовать совершенно другие цифры. Таким образом, фразы позволяют безболезненно мигрировать на другие кристаллы (с известными оговорками, т.к. у разных кристаллов может быть свой набор конфигурационных битов, но имена фраз по функциям будут совпадать).

Таким образом, если вместо

```
#include <pic.h>
__CONFIG (INTIO & UNPROTECT & LVPDIS & BOREN & MCLRDIS & PWRTEN &
WDTDIS);
```

мы запишем только `__CONFIG (0x3FFC & 0x3FFF & 0x3F7F & 0x3FFF` & и т.д. то и эта запись будет корректно скомпилирована.
Последовательность перечисляемых фраз или их чисел не имеет значения.

Кратко на русском перечислим выбранные биты конфигурации

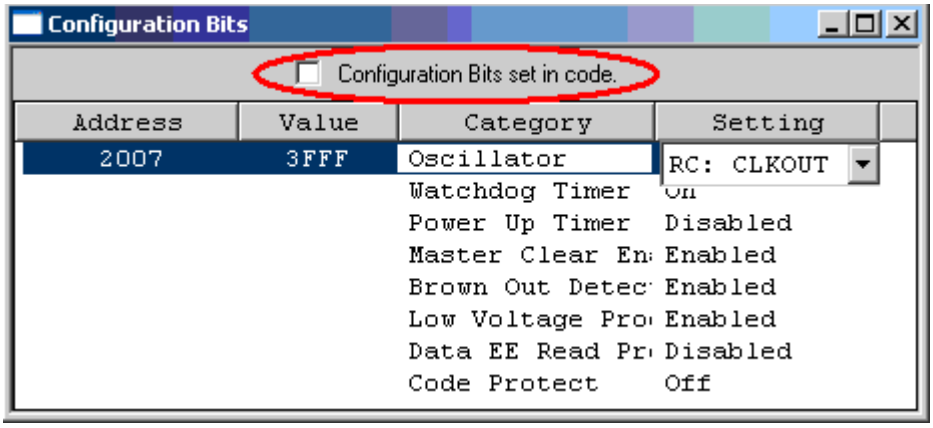
`INTIO & UNPROTECT & LVPDIS & BOREN & MCLRDIS & PWRTEN & WDTDIS`

`INTIO` внутренний генератор, RA6 работает как цифровая линия ввода/вывода;
`UNPROTECT` защита отключена;
`LVPDIS` отключено низковольтное программирование;
`BOREN` включен сброс по снижению питания;

`MCLRDIS` отключен внешний мастер сброса;
`PWRTEN` включен таймер включения питания;
`WDTDIS` отключен сторожевой таймер.

Что это значит и для чего – читаем даташит PIC16F627_628.pdf стр.88.

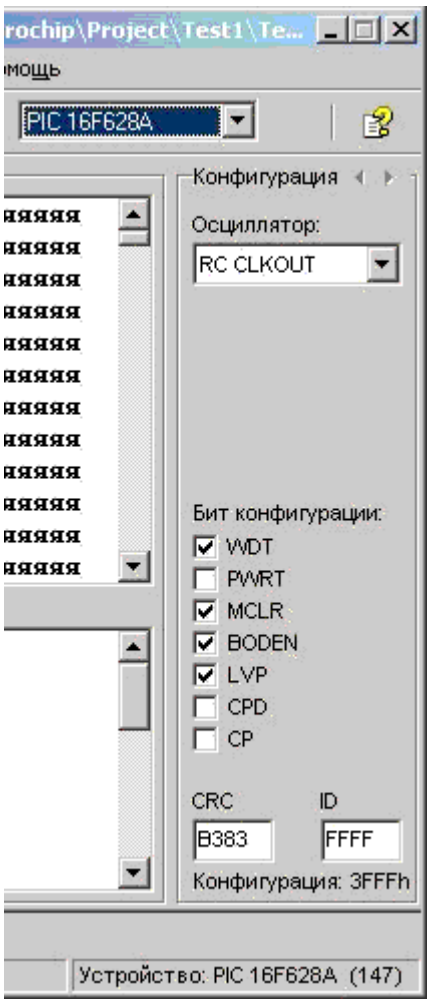
Необходимо отметить, что биты конфигурации интегрируются в прошивку (у микроконтроллеров AVRэто приходится делать вручную весьма запутанной установкой фьюзов). Если мы не конфигурируем МК, то будут установлены биты конфигурации по-умолчанию. Умолчания описаны в даташите. Но мы можем посмотреть в программе эти умолчания через меню **Configure** - **Configuration Bits…**



Сняв галочку, можно переопределить конфигурацию через вполне внятные обозначения. Установив галочку, настройки конфигурации будут интегрированы в прошивку. Это относится к случаю, если строка конфигурирования отсутствует в тексте. Приоритет по выбору битов конфигурации отдаётся строке конфигурации, а не этому окну настройки.

Рекомендую здесь же зайти в меню **Configure** – **Select Device…** где можно выбрать другой кристалл. Вам это может пригодиться, если вы захотите партировать (перенести) этот проект на другой кристалл, например, на PIC16F628 (без индекса A), или любой другой подходящий по количеству ножек и функций. Но это частный случай. Просто примите на заметку.

Я категорически не рекомендую надеяться на любые умолчания, описанные в даташите. Все параметры конфигурации, которые только можно указать – указывайте. Если по каким-то причинам вы захотите изменить биты конфигурации для имеющейся прошивки, то в программе для прошивания (например, в айси-проге IC-Prog) вы можем видеть нечто подобное (см. ниже). Лучше один раз правильно прописать биты конфигурации в исходнике, чем каждый раз напрягать мозг и вручную проставлять галочки в «швейной» программе.



Чуть выше мы помянули RA6. У вас возник вопрос – что такое RA6. Это одна из цифровых линий МК. На стр.3 даташита картинки с мнемоническими обозначениями линий. RA6 , означает «шестая линия порта А». О портах вы прочтете чуть далее.

Не менее важным элементом в строчке конфигурации является символ точки запятой «;» в конце предложения. Без него не компилируется. Этот символ является указателем конца. Будьте внимательны.

Наконец мы с вами добрались до нашей первой, но самой важной функции под названием main (англ. – «основа»).

```
void main (void)
{
}
```

Что такое функция? Это совокупность команд, обычно предназначенных для решения определенной задачи.

Физиололгия работы программы в МК.

Вы представляете **как работает МК?** Я вам расскажу. Всё довольно просто. Подаём питание на МК. Счетчик команд сбрасывается в ноль и зачитывает первую команду. Затем счетчик увеличивается на единицу (инкрементируется) и МК зачитывает следующую команду. Все эти команды физически записаны во флеш-память

программ. И так они друг за дружкой и зачитываются.

А как, глядя на исходник программы понять, в какой последовательности команды будут выполняться? И здесь всё просто. Первой командой в исходнике является первая команда в функции main *. И так далее, последовательно друг за дружкой, команда за командой выполняются команды в функции main. Зачитав последнюю команду в функции main, а фактически «споткнувшись» об закрывающуюся фигурную скобку, программа делает переход к первой команде функции main и таким образом программа закольцовывается.

*(на уровне физиологии и в терминах ассемблера это не совсем верное высказывание)

Вы должны понять, что работа МК никогда не останавливается, МК всегда бьётся и тактируется и всегда что-то делает в явном или неявном виде. Справедливости ради следует сказать, что есть частный случай спящего режима, когда мы МК «отправляем в спячку» для экономии энергии.

Итак, внутри фигурных скобок располагаются команды, которые выполняются друг за другом, с первой до последней, по кругу. Как и в какой последовательности расположить текст программы во флеше решает компилятор в соответствии с текстом нашего исходника. Результатом работы компилятора являются машинные коды в hex-файле.

В нашем примере в фигурных скобках пусто. Никаких команд нет. И именно эту программу мы называем «Рыбой». Так или иначе hex-файл создается в котором компилятор уже делает некоторые записи по конфигурированию МК и организации пустого цикла; напоминаем – hex-файл создается и располагается в папке с проектом.

Как работают функции. Комментарии.

У вас должен возникнуть закономерный вопрос – почему по-разному записаны функции, хотя я и говорю что это одно и то же:
`void main (void){}` и `main (){}` ?
Что такое `void` ? Для чего нужен `void` или не нужен совсем?

Вспомните что такое функция. Функция – это совокупность команд, обычно предназначенных для решения определенной задачи. А какой или каких задач? Любых (шучу). Любых задач связанных с обсчетом данных (чисел). Соответственно, мы можем функции передавать данные для обсчета, можем получать обсчитанные данные, можем и передавать и получать, а можем ничего не передавать и ничего не получать. Иначе говорят так – «передаем значение функции» или «функция возвращает значение».

Теперь переведем слово `void` – с англ. «пустое». Таким образом, функция у нас звучит «пустое майн пустое» или даже так «пустое основа пустое». Т.е. функция ничего не возвращает, и в функцию мы ничего не передаем.

Теперь рассмотрим некую функцию, которая называется «sum»:

```
unsigned int sum (unsigned int x, unsigned char y)
```

`unsigned int` перед sum – означает, что функция возвращает значение типа «целое без знака».

`unsigned int x, unsigned char y` – означает что в функцию мы передаем значение типа «целое без знака» которое равно x, а также передаем значение типа «символ без знакат» которое равно y.

Важный вывод. Фраза до фразы `sum` показывает какой тип данных будет возвращен функцией. Необходимо понимать, функция возвратит только одно значение (одно число). Необходимо также понимать, что функция не может возвращать одновременно больше одного значения. Либо одно значение одного из типов, либо ничего (пустое), т.е. void.

Фразы в скобках после sum показывают какого типа данные будут переданы в функцию. Здесь ограничений нет, в функцию можно передавать много значений разных типов.

Как бы мы не назвали функцию (хоть zadnica) мы обязательно должны сказать, что передаем в функцию и что по результатам вычислений функция возвращает, ну например, void zadnica (long int her).

Чтобы понять, как это работает, напишем маленький текст программы.

```
// объявляем функции
unsigned int sum (unsigned int x, unsigned char
y); // суммирование
void del (void); // деление

// объявляем глобальные переменные
unsigned int z;

// описание функции суммирования
unsigned int sum (unsigned int x, unsigned char y)
{
return x+y;
}

// начало программы
void main (void){
unsigned int q; // локальная переменная q
q = sum (5,1); /* вызываем функцию sum и передаем ей значения 5 и
1,
результат вызова sum (5,1) передаем q */
z = q; // значение из q передаем z
del();} // вызываем функцию деления

// описание функции деления
void del (void)
{
```

```
z = z/2; // результат деления z/2 передаем снова в z
}
```

Заголовочный файл и строчку конфигурации я не стал писать в этом примере. Вы уже должны понимать о чем я говорю. Ну а теперь давайте разбирать. На всё это мы будем смотреть глазами компилятора (т.е. глазами переводчика программы с языка Си в машинные коды).

В тексте программы мы видим комментария. Комментарии начинаются с символов // и распространяются в пределах одной строчки. Также комментария могут быть многострочными. Текст, который стоит между /* */ будет закомментирован. Эти символы (/**/) удобно набирать в дополнительной цифровой секции клавиатуры (рядом с кнопкой NumLock). Компилятор игнорирует комментария и пустые строчки.

Для того, чтобы функция могла быть **использована** в программе – её нужно **объявить** и **описАть**. Объявление – это заголовок функции, например, void primer (void); . А описание, это – это заголовок с телом функции (под телом понимаются команды внутри фигурных скобок).

Необходимо отметить, что объявления одних функций и описания других функций могут в исходнике могут существовать сами по себе, если они в программе не используются. Да, бывает и такое. Что-то написали, а фактически в программе не используется. Да, всё успешно компилируется. Да всё прекрасно работает. В этот абзац можно не вникать. Со временем поймете всё на практике.

Объявление и описание функции не является обязательным правилом для функции main , т.к. мы знаем, что **функция main обязательно должна быть в тексте. И по логике работы микроконтроллера ни кто не может вызвать функцию main и в функцию main ничего не передается**. Отсюда и послабления в оформлении текста программы: функция main может не описываться; также до и после её названия могут не указываются типы данных. Таким образом, void main (void) соответствует main () (круглые скобки после main() строго нужны).

Объявления и описания функций нужны, чтобы функция main могла вызвать и использовать объявленные и описанные функции. Под вызовом и использованием понимается выполнение команд сгруппированных в вызываемой функции.

Обратите внимание на то, в какой последовательности расположены функции в программе. Функции расположены в произвольном порядке. Это отдельные программные конструкции. Не важно в каком порядке стоят функции. Ставьте как вам удобно.

Рассмотрим функцию `main`

```
// начало программы
void main (void){
unsigned int q; // локальная переменная q
q = sum (5,1); /* вызываем функцию sum и передаем ей значения 5 и
1,
результат вызова sum (5,1) передаем q */
```

```
z = q; // значение из q передаем z
del();} // вызываем функцию деления
```

Первая строчка после фигурной скобки `unsigned int q;` – некая локальная переменная «кю». Нам эта переменная требуется для временного хранения результатов расчетов внутри функции. После выполнения функции эта переменная «разрушается». Физически под эту переменную компилятор выделит ячейки памяти в оперативной (регистровой) памяти. **Локальная переменная или несколько локальных переменных ставятся в самом начале тела функции.**

Обратите внимание. Фигурные скобки лишь очерчивают границы функции. Вы можете располагать фигурные скобки как вам удобно для зрительного восприятия. Также каждая командная конструкция заканчивается знаком точки с запятой «;». Ни кто не мешает записать всю функцию в одну строчку, которая будет успешно компилирована.

```
void main(void){unsigned int q;q=sum(5,6);z=q;del();}
```

Что такое `q = sum (5,1);`? В этом месте программы будет вызвана функция `sum`, которой будет передано два значения 5 и 1. Необходимо понимать, что объявление вызываемой функции должно по тексту программы находиться раньше, чем будет сделан вызов функции. Объявления функций, это лишь «уведомления» для компилятора, что рано или поздно или никогда будет вызвана функция. Итак, вызов состоялся. Дальше компилятор перемещается в текст функции sum.

```
unsigned int sum (unsigned int x, unsigned char y)
{
return x+y;
}
```

А в функции sum всего лишь одна строчка: сложить x и y и вернуть результат (return). Функция sum не знает, кто заберет результат вычислений. Кто вызовет, тот и получит. А вызывает у нас строчка

```
q = sum (5,1);
```

В итоге в q у нас будет сумма двух чисел 5 и 1. Обратите внимание какие типы значений у нас заявлены в функции. Складывается беззнаковый инт и беззнаковый чар. Результат вычисления передается в беззнаковый инт. Т.е. величина складываемых чисел допустима к выбранному типу данных. Однако, для экономии программного кода, следует правильно продумывать размерность обсчитываемых чисел и выбор соответствующего типа данных. В нашем примере эта нерациональность сделана для наглядности.

Итак, в q у нас сидит число 6.

Следующая строчка тупо передает значение из q в z . Теперь у нас в z сидит число 6. Можно было бы сразу написать z = sum (5,1); но мы не ищем легких путей. Физически под эту переменную компилятор выделит ячейки памяти в оперативной (регистровой) памяти, и эти ячейки будут зарезервированы всё время за z, пока на кристалл идет питание.

В тексте программы мы объявили так называемые глобальные переменные. Что это значит? А вы чувствуете интуитивно разницу между локальными и глобальными переменными? Глобальные переменные могут использоваться всеми функциями. А вот локальные переменные, только внутри соответствующей функции и потом они «разрушаются». Освободившаяся оперативная память используется другими локальными переменными. Для чего я это говорю. Экономнее ребята, экономнее. Оперативной памяти не так много. Поэтому разумно используйте глобальные переменные.

Имена любых переменных обязательно начинаются с символов латинских букв, а не с цифр; буквы верхнего и нижнего регистров это разные буквы; имена переменных не могут совпадать с именами функций. У глобальных переменных всегда уникальные имена, которые не совпадают с именами локальных переменных. Локальные переменные в разных функциях могут иметь одинаковые имена, чего я и придерживаюсь – во всех локальных переменных задаю имя tmp (перечитайте этот абзац) (перечитайте дважды).

del();} // вызываем функцию деления
В эту функцию мы ничего не передаем и она ничего не возвращает. Тогда зачем она? Если заглянуть в эту функцию, то можно увидеть, что она работает с глобальными переменными. Точнее с одной переменной z.

```
void del (void)
{
z = z/2; // результат деления z/2 передаем снова в z
}
```

Т.к. ранее мы в z поместили число 6, то в результате вызова этой функции будет произведено деление 6/2 и получено число 3, которое затем будет записано назад в глобальную переменную z.

Важные выводы.
1) Функция может вызывать другую функцию.
2) В функцию для обсчета могут передаваться несколько значений.
3) Результатом вызова функции может стать возврат строго одного значения для некой переменной.
4) Функция обсчитывает глобальные и локальные переменные.
5) Прежде чем вызвать функцию, её нужно объявить и описать.
6) Описания функций могут находиться в любом месте программы и чередоваться как угодно
7) Объявление функции должно находиться перед вызовом функции.
8) Глобальные переменные объявляются перед их использованием.
9) Глобальные переменные имеют уникальные имена.
10) Локальные переменные объявляются в теле функции в самом начале функции.
11) Имена локальных переменных в разных функциях могут совпадать.
12) Тело функции – это текст в фигурных скобках.
13) Комментарии пишутся после // , либо между /* */.
14) Функции могут быть написаны в одну строчку.

15) Функции имеют уникальные имена.
16) Имена не должны начинаться с цифр.
17) В дерево проекта можно подключать иные файлы-исходники, но только один из файлов должен содержать функцию main, т.е. компилятор ищет не файл с именем main.c , а функцию с таким именем в подключенных файлах.

Программа №2. «Продолжение рыбы». Именование портов и линий

Ну что за шутки, скажете вы, прочитав этот заголовок. На самом деле от вас требуется еще немного терпения, чтобы сделать первый практический шаг. Я не стал напрягать вас раньше времени, иначе в вашей голове были бы каша, мёд, гавно и пчелы. Чтобы этого избежать, требуется системность изложения.

Создайте проект «test2» и поместите в него программу ниже.

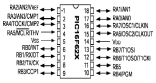
```
#include <pic.h>
//pic16f628a
/* тестовая программа № 2 из самомучителя*/

__CONFIG (INTIO & UNPROTECT & LVPDIS & BOREN & MCLRDIS & PWRTEN &
WDTDIS);

#define knopka RB4 // кнопка
#define diod   RA0 // светодиод

void podgot (void); // подготовка МК

void main (void)
{
podgot();
diod = knopka;
}

// === подготовка МК
void podgot (void)
{
TRISA = 0b00000000; // направление работы ножек порта А
TRISB = 0b00010000; // направление работы ножек порта В
CMCON = 0x07; // отключение компараторов
PORTA = 0; // очищаем порт А
PORTB = 0; // очищаем порт Б
RBPU = 0;  // подтягивающие R (0-вкл, 1-выкл)
}
```

Рассмотрим этот текст. Мы подключили заголовочный файл и сконфигурировали МК. Я обычно в самом верху добавляю комментарий о том, под какой кристалл «заточена» конфигурация и проект; при желании пишу еще несколько слов о программе, чтобы потом не чесать репу.

```
#define knopka RB4 // кнопка
#define diod   RA0 // светодиод
```

Каждая из этих строчек дает компилятору указание, что под одной фразой должна пониматься другая фраза. Например, если в тексте программы будет встречена фраза knopka , то это он должен понимать как RB4. Или если в тексте встретится фраза diod , то это RA4.

Мы ранее говорили, что RAx и RBx это цифровые линии МК. Под «**x**» (икс) понимается порядковый номер цифровой линии в МК. Порядковый номер может быть от 0 до 7, т.е. восемь бит. Ножки как и у любого DIP-корпуса микросхемы нумеруются против часовой стрелки, начиная от ключа, если смотреть на корпус сверху (со стороны маркировки). Для чего я говорю очевидные вещи? Будьте внимательны, номера ножек и номера цифровых линий портов разные понятия. Соответствующей цифровой линии порта соответствует определенная физическая ножка. Давайте рассмотрим рисунок со стр. 3 даташита.



Как видим RB4 соответствует вывод 10, а RA0 – 17й вывод. Так или иначе, по контексту вы будете понимать о чем идет речь – о номере электрического вывода или об имени цифровой линии.

На картинке вывод 5 – это минусовая линия питания Vss. Вывод 14 – плюсовая линия питания Vdd. Питаемся мы стабилизированным напряжением 5 вольт, для чего достаточно интегрального стабилизатора 78L05 с обвязкой с двух сторон по 10 мкф (кто не в теме по 78L05 и аналогичных стабилизаторах, по-быстрому читаем документацию нашего сайта).

Также на картинке видим разнонаправленные, стрелки которые нам говорят о том, как может работать ножка – на вход/на выход (понятно, что линии питания только на вход, т.е. входящие сигналы). Большая часть ножек работает и на вход и на выход. RA5 в данном кристалле работает только на вход.

Кроме RAx и RBx мы видим и другие обозначения через дробь. Это информация о выводах встроенных аппаратных модулей и специальных выводах, обеспечивающих определенные режимы работы МК. На данном этапе обучения нас интересуют только RAx и RBx.

Полагаю, что вы уже получили первичные сведения о цифровых линиях на портах из даташита со стр. 27. Давайте кратко повторим.

– цифровые линии МК сгруппированы в так называемые порты;
– порты именуются буквами латинского алфавита (порт A, порт B и т.д.)
– каждый порт имеет до 8 цифровых линий;
– линии нумеруются от 0 до 7 (например, RB0… RB7);
– линии на портах работают в режиме цифрового ввода/вывода;
– линии могут быть программно переключены на встроенные модули МК, выполняющие определенные задачи;
– линии порта B имеют программно подключаемые подтягивающие резисторы (об этом подробнее чуть позже);
– линия RA4 может формировать только НЛУ,

– линия RA5 работает только на вход
(на стр. 30 даташита заголовок рисунка с опечаткой – правильно читать «**RA5**/MCLR/THV») – ножки RA4 и RA5 я называю «цифровыми костратами», но у них есть офигенные функции.

Для чего нужны эти сведения? Благодаря этим знаниям вы наиболее оптимальным образом будете разводить печатные платы, а под имеющиеся электрические соединения писать программу.

А теперь глазками снова посмотрим текст нашей программы. Функция майн. В ней первой строчкой вызывается некая функция podgot, а затем идет некое равенство.

```
void main (void)
{
podgot();
diod = knopka;
}
```

Давайте рассмотрим первую строчку, которой вызывается функция podgot. Перемещаем глазки в функцию podgot и изучаем её содержимое.

Мы знаем, что ножки могут работать на вход или на выход. Чтобы четко определить направление работы, необходим ряд манипуляций. За направление работы отвечают регистры TRISx. В даташите они упоминаются. Говоря по-русски, в эти регистры нужно прописать число, где каждый бит этого числа определяет то или иное направление той или иной ножки.

Регистры портов. Определение направлений работы линий.

Что такое регистр? Регистр – это ячейка памяти в МК (в области ОЗУ). Если это ячейка памяти, то в неё можно записать один байт информации. А байт это некое число, которое лежит в десятичном диапазоне 0…255. Некоторые регистры/ячейки имеются определенные имена и отвечают за тот или иной режим работы МК, что очень подробно расписывается в даташите.

Рассмотрим строчку `TRISB = 0b00010000;`

Мы видим, что регистр TRISB приравнивается чему-то. Вы наверняка слышали о некоторых системах счисления: двоичная, десятичная, шестнадцатеричная. И чтобы компилятор понимал, какое число в данном случае употребляем, существуют определенные форматы записи.

```
127 – обычное десятичное число 127;
0x7F – число 127 в 16-ричной системе счисления;
0b01111111 – число 127 в двоичной системе счисления.
```

Если нет приставки – то это десятичное число, если префикс 0x – то это 16-ричное число, если префикс 0b – то это двоичное число. Что такое двоичные и 16-ричные числа – напрягаем Интернет. **Программу для перевода двоичных, 16-ричных и десятичных чисел** скачиваем здесь, либо воспользуйтесь стандартным калькулятором Windows (вид – инженерный).

Почему мы TRISB приравниваем некому двоичному числу? Для ответа на этот вопрос сначала заучим правило: **биты в байте нумеруются справа налево от нуля до семи.** Каждый бит в регистре/ячейке TRISB отвечает за направление работы соответствующей ножки на порту B. Например, нулевой бит отвечает за направление работы ножки RB0. Фраза TRISB = 0b00010000; говорит нам, что четвертый бит в этом регистре будет равен 1. Использование двоичной записи гораздо нагляднее показывает направление работы каждой ножки.

А как запомнить – какой бит (0 или 1) нужен для работы на вход или для работы на выход? Мы говорили, что ножки работают на вход или на выход (вход/выход – на англ. Input/Output или IO – наверняка вы это слышали в аббревиатуре B**IO**S– базовая система ввода/вывода). Внешне буквы I/O похожи на цифры 1/0. Цифра 1 – это вход (Input), а цифра 0 – это выход (Output).

Итак, гладя на фразу TRISB = 0b00010000; мы видим что 4й бит равен единице. А это значит что RB4 будет input, т.е. цифровой линией, работающей на вход. Если бы мы записали TRISB = 0b00100010; то этим мы скажем, чтоRB0-RB2-RB3-RB4-RB6-RB7 работают на выход, RB1 и RB5 на вход.

```
TRISA = 0b00000000; // направление работы ножек порта A
TRISB = 0b00010000; // направление работы ножек порта B
```

Те кто не понял смыл этих двух строчек, перечитайте несколько абзацев выше.

Следующая строчка CMCON = 0x07; // отключение компараторов. Для тех кто не в теме объясняю – в МК типа PIC16F62Xвстроен модуль компараторов. Выводы этого модуля имеют названия AN0…AN3 (см. картинку на стр. 3 даташита). По-умолчанию (после сброса питания МК) ножки RA0…RA4 не являются цифровыми линиями ввода/вывода; они подключены к модулю компараторов. Нам нужно сделать переключение с AN0…AN3 на RA0…RA4. Именно за это отвечает строчка CMCON = 0x07;. Более подробнее смотрите даташит стр. 54 (рекомендую не смотреть).

Далее у нас две строчки

```
PORTA = 0; // очищаем порт A
PORTB = 0; // очищаем порт Б
```

Для чего нужна очистка? Представим ситуацию – мы подаем питание на МК. Начинается выполнение программы. Выполнились строчки определения направления. И что после этого? На ножках, которые работают на выход может быть любое состояние сигнала. Признаком хорошего тона является именно в этом месте программы, в самом начале установить на ножках НЛУ (если обратного не требует логика работы устройства). Это как минимум экономит энергию, а как максимум даёт вам уверенность в определенности состояния сигналов на ножках.

Что такое очистка? Это прописывание нуля в регистр или установка конкретного бита в ноль. Эти две строчки можно было бы написать и в двоичном формате

```
PORTA = 0b00000000; // очищаем порт A
PORTB = 0b00000000; // очищаем порт Б
```

Те же яйца, вид с боку. Это означает, на всех цифровых линиях, **которые работают на выход,** будет установлен НЛУ. Если бы мы записали PORTB = 0b10010001 то на RB0, RB4 и RB7 будет установлен ВЛУ (при условии, если эти ножки будут настроены на выход).

Внимание. Те кто не понял, читайте заново и перечитывайте, пока не уловите логику. Это основы основ – определить направление работы ножек и установить соответствующий уровень сигнала на соответствующей ножке.

Теперь рассмотрим работу с конкретными битами регистров TRISx и PORTx (это позволит работать индивидуально с ножками, а не со всеми сразу). По пути C:\Program Files\HI-TECH Software\PICC\9.50\include\ в известном вам файлеpic16f62xa.h приводятся наименования регистров и битов в регистрах. В файле мы можем найти фразы TRISA, TRISB,PORTA, PORTB – это названия регистров (TRISx отвечают за направление работы, PORTx используются для установки исходящих сигналов на линиях и оценки входящих сигналов). Именно в эти регистры в тексте программы мы прописываем некоторые числа. Однако мы видим индивидуальные биты, например TRISA0, TRISB1, RA2, RB3 и прочие.

Фразу TRISB = 0b00010000; можно было бы заменить набором фраз

```
TRISB0=0; TRISB2=0; TRISB5=0; TRISB7=0;
TRISB1=0; TRISB3=0; TRISB4=1; TRISB6=0;
// аналогично определяются направления через TRISAx, где x – соотв. бит
```

и последовательность фраз не имеет значения. В общем мы имеем 8 фраз, в которых мы «поклонились» каждому биту, т.е. определили **направление** работы каждой ножки. Ну а теперь давайте посмотрим как можно индивидуально установить сигнал на ножке. Заменим фразу PORTB = 0b01011011; набором фраз

```
RB0=1; RB2=0; RB5=0; RB7=0; RB1=1; RB3=1; RB4=1; RB6=1;
// аналогично определяется уровень сигнала через RAx, где x – соотв.бит
```

Таким образом, установили определенный уровень сигнала на конкретной ножке. И вы должны четко понимать, что PORTB= 0b01011011 и PORTB = 91 это одно и то же, но менее наглядно. И теперь вы должны понять, почему двоичная запись для таких случаев более удобна; мне проще написать двоичное число, посчитав количество битов и комбинацию нулей и единиц, чем переводить двоичное число в десятичное или 16-ричное.

Всё, хватит про порты. Неохваченным остался вопрос работы с ножками, которые настроены на вход. Об этом чуть позже.

Риторическое отступление. Те, кто переходит с Ассемблера на Си, последние две страницы прочитал с легкостью и радостными чувствами. Действительно, удобность управления портами и ножками поражает своей простотой и наглядностью. На ассемблере это выглядит менее понятно и менее удобно. И вы уже должны

интуитивно предположить, как работать с ножками, настроенными на вход. Конец отступления.

Последняя строчка в функции podgot звучит как

```
RBPU = 0;  // подтягивающие R (0-вкл, 1-выкл)
```

R – это символ резистора (сопротивления). Что такое подтягивающие резисторы? Мы знаем, что МК работает с сигналами ВЛУ и НЛУ, то есть с нулями и единицами. А теперь представим, что ножка настроена на вход и «висит в воздухе» (ни к чему не подключена). Какой уровень сигнала на ней в таком случае будет? Мне трудно сказать определенно. И я не далек от истины. Действительно, на ножке будет неопределенное состояние, вызываемое наводками, что приводит к хаотичному появлению ВЛУ и НЛУ на ножке. И это означает, что на ножке так называемое серое состояние и его иначе называют третьим состоянием. Чтобы избежать подобной неопределенности, ножки «подтягивают» через сопротивления к плюсу или к минусу питания (традиционно к плюсу).

Что такое подтягивание? Это подключение сопротивления (резистора) одной стороной к цифровой линии МК, другой стороной к линии питания МК. Номинал этого сопротивления принципиального значения не имеет и может находиться в пределах 1 ком – 10 ком (я ставлю 4,7-5,1 ком).

Все известные мне PIC имеют на порту B программно включаемые подтягивающие резисторы к плюсу. В данном случае резисторы включаются и отключаются «скопом»: либо подтягивающие резисторы (8 шт.) подключены ко всем линиям порта B, либо ни одна линия не подтянута. За подключение/отключение отвечает специальный бит, который называется RBPU. Это, пожалуй, единственный инвертированный бит. Если мы его приравняем нулю, то значит подключим подтягивающие резисторы. И наоборот, прописав в этот бит единицу, мы отключим подтягивающие резисторы.

Наконец мы с вами рассмотрели 6 строчек функции podgot. Это ни что иное, как продолжение нашей рыбы. Ни одна программа не обходится без подготовки ножек портов. Можно конечно сделать хитрый проект и использовать состояние регистров TRISx и PORTx по-умолчанию в МК, т.е. использовать состояние битов после сброса (это также отражено в даташите). Но береженого бог бережет. Ни на что не надейтесь. Излишняя уверенность – повод для ошибок. Запомните –**если что-то не работает – проблема либо в программе, либо в электрических соединениях** (99,999%). Ну произошел провал напряжения, ну сбросился МК, а какие-то ячейки не сбросились в умолчания, вот вам и траблы (баги, глюки).

Строго говоря, направление работы ножек можно в любой момент переопределить в программе, в зависимости от поставленных целей и внешней схемотехники. А изменение уровня сигнала на линиях – это обычное дело; т.е. мы дергаем ножками. В наших первых учебных проектах мы будем строго определять направления работы ножек и уровень сигналов на линиях в самой первой вызываемой функции podgot (функцию можете назвать иначе, хоть zopa).

Ну что друзья, переходим к практике. Не бздите, всё довольно просто.

Мы помним что есть такие записи

```
#define knopka RB4 // кнопка
#define diod   RA0 // светодиод
```

В функции main есть строчка

diod = knopka;

что означает «RA0 присвоить состояние из RB4». Смотрим схему.

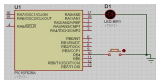


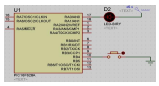
Схема нарисована в Протеусе (Proteus). В этой схеме предполагается, что питание подано на соответствующие линии питания (их нет на картинке).

А теперь разбираем работу схемы.
1) Внутри МК мы включили подтягивающие резисторы (у нас это делается только на порту Б и только к плюсу питания), таким образом у нас на RB4 ВЛУ или иначе говоря единица до тех пор, пока не нажата кнопка. Если мы нажимаем кнопку, то мы закорачиваем RB4 на минус питания, что переводит ножку в НЛУ (в ноль).
2) Программа у нас зациклена в main. RA0 присваивается состояние которое на RB4. Когда кнопка не нажата, на RA0 у нас единица, т.е. ВЛУ и светодиод светится. Нажимаем кнопку, и светодиод гаснет, т.к. на RA0 устанавливается ноль.

Т.к. цикл крутится с большой частотой (при частоте 4 мГц МК выполняет 1 млн операций в секунду) кнопка у нас опрашивается ну очень часто. Это приводит к моментальному реагированию светодиода на нажатие кнопки.

Именно так оценивается состояние сигнала на ножке, которая работает на вход – оцениваются входящие сигналы.

Вам не нравится такой алгоритм управления? Сделайте иное включение

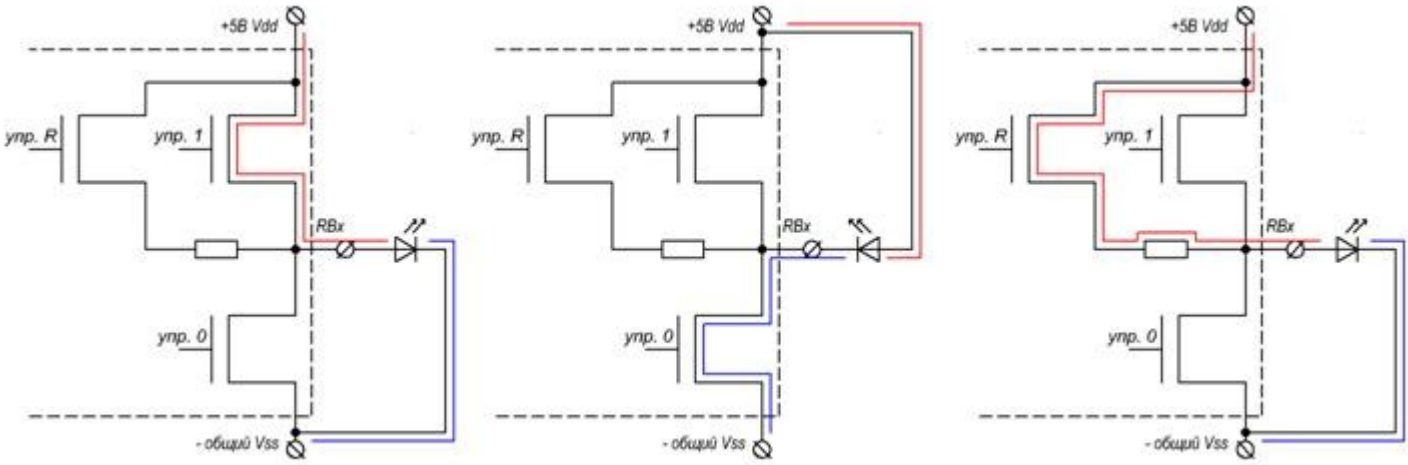


Было бы хорошо, если бы вы заметили дефекты в программе. Их два.
1) Т.к. программа крутится в цикле, то у нас постоянно вызывается функция podgot и выполняются действия, которые нам нужно сделать только единожды. Переопределение направления работы ножек (пусть даже и без изменения направления) приводит к тому, что ножка в момент определения направления может находиться а неопределенном состоянии, а это фактор нестабильности.
2) В цикле мы постоянно чистим порты. С одной стороны мы устанавливаем RA0=RB4 и ожидаем, что оставшаяся часть цикла программы не будет затрагивать ножку RA0 на порте А. У нас же происходит обратное. Мы установили сигнал, и практически сразу же у нас он снимается чисткой. Это приводит к мерцанию диода с высокой (незаметной глазу) частотой. Мы можем закомментировать //PORTA = 0; но это не выход из ситуации.

И что делать? Напишите функцию main иначе (см. ниже). Нововведения мы рассмотрим в следующем разделе.

```
void main (void)
{ // начало функции
podgot(); // вызов функции подготовки МК
while(1)
{ // начало бесконечного цикла
diod = knopka;
} // конец бесконечного цикла
} // конец функции
```

А под конец этого раздела мы посмотрим работу со светодиодом с точки зрения физиологии. Смотрим картинки ниже.



В даташите вы видели структурные схемы. Это их небольшие фрагменты. Здесь я показываю, как течет ток в той или иной ситуации. Здесь три картинки.
1) Ножка RBx работает на выход (подтягивающий резистор автоматически отключен). Прописывая RBx=1 мы устанавливаем сигнал высокого логического уровня (ВЛУ), т.е. напряжение равное напряжению питания.
2) Ножка RBx работает на выход (подтягивающий резистор автоматически отключен). Прописывая RBx=0 мы устанавливаем сигнал низкого логического уровня (НЛУ), т.е. напряжение равное 0 вольт.
3) Нетрадиционный способ управления. Ножка RBx работает **на вход**. Прописывая RBPU = 0 или RBPU = 1 мы включаем или отключаем подтягивающий резистор на RBx, т.е. устанавливаем ВЛУ. И этот вариант имеет право на жизнь.

Ифы, форы, вайлы или основы интеллекта. Истина не ложь

Бытует мнение, что для приличной игры на гитаре достаточно знать три аккорда и иметь две струны. Не знаю, правда это или нет, но мы с вами рассмотрим три оператора, из которых вы будете творить свои шедевры.

Озвучив в заголовке громкое слово «интеллект», нам необходимо это понятие перенести на МК. Правильнее этот термин называть логикой или логической последовательностью действий. Ну а что такое логика, спросит дотошный читатель? В ответ на это вспоминается анекдот про мужиков, которые играют во дворе в

домино. Один из мужиков приводит интересную логическую последовательность: креветки – пиво – супруга – мало – 100 гр водки – член – стоит. А другой мужик эту логическую последовательность сокращает: ты не любишь креветки, значит у тебя не стоит (уж позвольте мне такие вольности изложения).

Логика в МК сродни логике в жизни, но гораздо однозначнее и определеннее. И, разумеется, все логические действия предопределены программой. Логика МК построена на сравнении – больше или меньше, равно или не равно, и комбинация этих условий. Хватит ля-ля, перейдем к делу.

Для начала поймем, что такое истина и ложь. Большинство команд строится с использованием операторов языка Си и условий истины и лжи. Операторы проще понимать как некие логические конструкции. Операторы в зависимости от истины и лжи работают определенным образом. Итак, под истиной понимаются правдивые высказывания, например,

примеры истинных высказываний	
5>2	больше
3<4	меньше
6==6	равно
5!=6	не равно
4<=5	меньше или равно
4>=3	больше или равно
1	единица – это всегда истина

очевидно, что под ложью понимают не правдивые высказывания

примеры ложных высказываний	
1>2	больше
5<4	меньше
5==6	равно
6!=6	не равно
6<=5	меньше или равно
2>=3	больше или равно
0	ноль – это всегда ложь

Два забавных примера:
ложь != истина – это истинное высказывание
ложь == истина – это ложное высказывание

В таблице мы с вами рассмотрели примеры на числах. В операторах на практике используются переменные либо переменные и числа.

if-else (если-иначе)

Синтаксис

```
if (выражение)
[группа операторов 1]
else
[группа операторов 2]
```

Если выражение истинно (т.е. то, что в круглых скобках), выполняется группа операторов 1 [т.е. то, что в квадратных скобках] (см. синтаксис). Если выражение ложно, выполняется группа операторов 2. В этом операторе **else** можно не использовать. В таком случае если выражение ложно, выполняются следующие операторы.

Пример из жизни
```
if (term > 45) // если температура больше 45
vent = 1;      // включить вентилятор
else           // иначе
vent = 0;      // выключить вентилятор
```

Пример из жизни
```
if (term > 45) // если температура больше 45
{
vent = 1;      // включить вентилятор
cond = 1;      // и включить кондиционер
}
else           // иначе
{
vent = 0;      // выключить вентилятор
cond = 0;      // и выключить кондиционер
}
```

Пример из жизни
```
if (dol > rub)  // если долларов больше чем рублей
go = 1;         // то гуляем
else            // иначе
{
go = 0;         // не гуляем
sex = 0;        // совсем не гуляем
}
```

Пример из жизни
```
if ((many < 45)|(rebenok!=1)) // если денег меньше 45
                 // ИЛИ нет одного ребенка
{
go = 0;         // не гуляем
sex = 0;        // совсем не гуляем
narko = 0;      // и думаем о жизни
}
```
В этом примере если выполняется **ХОТЯ БЫ ОДНО** условие, то выполняется последующее выражение или группа выражений в фигурных скобках.
Пояснение. `rebenok!=1` имеется ввиду, что число детей не равно 1. В жизни число детей может быть равно нулю, а может быть равно двум, трём… Почувствуйте разницу, нет одного `(!=1)` и ни одного `(<1 или ==0)`.

Пример из жизни
```
if ((many > 200)&(auto>=1)&(blondinko==1))
gold = 1;
// если денег больше 200 И хотя бы одна машина И верная девушка
// то мы покупаем стринги счастья
```

Всё довольно просто и логично. Если соблюдаются **ВСЕ** условия, то выполняется последующее выражение или группа выражений в фигурных скобках. К любому условию можно подключить operatorelse , благодаря которому при не соблюдении условий можно выполнить иные выражения.

| - логическое ИЛИ
& - логическое И

Забавный пример: `((быть)|(!=быть))=?`

Также можно делать вложенные проверки

```
if (many>200)   // если денег > 200
{
if (auto == 1)  // если есть авто
more = 1;       // едим на море
else dacha = 1; // иначе едим на дачу
}
else       // иначе если денег мало
more = 0;  // нет моря
dacha = 0; // нет дачи
```

Таким образом, можно сделать множество разных проверок и их комбинаций, что приводит к ветвлению логики работы программы.

for (в течение)

Синтаксис

```
for (выражение1; выражение2; выражение3) тело;
```

Оператор **for** – это наиболее общий способ организации цикла.

Выражение 1 обычно используется для установления начального значения параметра цикла. Выражение 2 – это выражение, определяющее условие, при котором тело цикла будет выполняться. Выражение 3 определяет закон изменения параметра цикла после каждого выполнения тела цикла.

При выполнении оператора **for** сначала вычисляется выражение1, затем выражение2. Если выражение2 отлично от нуля (истина), выполняется тело цикла, вычисляется выражение3, снова вычисляется выражение2, и если оно по-прежнему отлично от нуля, то цикл повторяется. Если выражение2 равно нулю (ложь), то управление передается на оператор, следующий за оператором **for**.

Важно то, что проверка условия всегда выполняется в начале цикла. Это значит, что тело цикла может ни разу не выполниться, если условие выполнения сразу будет ложным.

Классический пример использования

```
for (tmp=0; tmp<10; tmp= tmp+1)
{
// тело цикла выполнится 10 раз, начиная
// со значения tmp=0 и закончится при tmp=9
}
```

Пример вычисления суммы чисел от 1 до 10

```
y = 0; // очищаем переменную под результат
for (x=1; x<=10; x=++)
{
y = y+x; // тело цикла
}
```

++ инкрементирование, т.е. увеличение на единицу
— декрементирование, т.е. уменьшение на единицу

Другим вариантом использования оператора **for** является бесконечный цикл. Для организации такого цикла можно использовать пустое условное выражение. Если отсутствует проверка, т.е. выражение2, то считается, что оно всегда истинно.

Пример бесконечного цикла с оператором **for**

```
for (;;)
{
// тело бесконечного цикла
}
```

while (пока)

Синтаксис

```
while (выражение) тело;
```

Оператор цикла **while** называется циклом с предусловием. В качестве выражения допускается использовать любое выражение языка Си, а в качестве тела – любой оператор, в том числе пустой или группа операторов в фигурных скобках.

Сначала вычисляется выражение; если оно ложно (или значение выражения равно нулю), то выполнение оператора **while** заканчивает и выполняется следующий по порядку оператор. Если выражение истинно (значение выражения отлично от нуля), то выполняется тело оператора **while** и процесс повторяется с начала.

Пример бесконечного цикла

```
while (1)
{
// тело цикла
}
```

Пример организации задержки

```
tmp = 0xffff;     // прописали некое число
while (tmp-->0); // декрементировать tmp и
        // выполнить пустое тело цикла
        // до тех пор пока >0
```

Пример цикла на 8 витков

```
tmp=0; // чистим переменную
while (tmp<8)
{
tmp++;
// тело цикла с некими операциями
}
```

do-while (делать пока)

Синтаксис

```
do тело while (выражение);
```

Оператор цикла **do-while** называется оператором циклом с постусловием и используется в тех случаях, когда

необходимо выполнить тело цикла хотя бы один раз.

Сначала выполняется тело цикла (которое может быть составным оператором), затем вычисляется выражение. Если оно истинно (значение выражение отлично от нуля), то тело цикла выполняется снова и т.д. Если выражение становится ложным (или равно нулю), то выполнение оператора **do-while**заканчивается и выполняется следующий по порядку оператор.

Пример цикла на 10 витков

```
tmp = 0; // чистим переменную
do{
tmp++; // инкрементируем
// еще что-нибудь делаем
} while (tmp<10); // проверяем условие
```

switch-case-break (выбрать набор и выйти)

Синтаксис

```
switch (выражение)
{
case константа1: группа операторов 1
case константа2: группа операторов 2
...
default: группа операторов n
}
```

Оператор **switch** предназначен для организации выбора из множества различных вариантов, который заключается в проверке совпадения значения данного *выражения* с одной из заданных *констант* и соответствующего ветвления.

Выражение, следующее за ключевым словом **switch** в круглых скобках, может быть любым выражением, допустимым в языке Си, значение которого должно быть целым. Значение этого выражения является ключевым для выбора из нескольких вариантов. Тело оператора **switch** состоит из нескольких операторов, помеченных ключевым словом case с последующими *константами*.

Если значение константы, стоящей после **case**, совпадает со значением выражения, следующего за ключевым словом **switch**, то выполнение начинается с этого варианта. Все константные выражения в операторе **switch** должны быть уникальны. Кроме операторов, помеченных ключевым словом **case**, может быть (обязательно один) фрагмент, но помеченный ключевым словом **default**.

Если ни один из вариантов не подходит, то выполняется оператор, стоящий после **default**. Префикс**default** является необязательным; если его нет и ни один из случаев не подходит, то управление передается на следующий после **switch** оператор. Варианты (**case**) и выбор по умолчанию (**default**) могут располагаться в любом порядке.

После выполнения операторов, соответствующих выбранному варианту, будут выполняться операторы, соответствующие следующему варианту. Для выхода из оператора **switch** используется оператор **break**.

Пример.

```
tmp = 2; // некая переменная
```

```
switch(tmp)
{ // начало тела switch
case 0: {tmp=tmp-2; tmp=tmp-2;}
case 3: tmp=tmp-2;
case 2: tmp=tmp*5;
case 5: tmp=tmp/2; break;
case 4: tmp=tmp+1; break;
default: ;
} // конец тела switch
```

Комментируем. Выполнение оператора **switch** начинается с оператора, помеченного **case 2**, т.к. tmp = 2. Таким образом переменная tmp получает значение tmp=2*5=10. Затем выполняется оператор, помеченный ключевым словом **case 5** (по порядку), tmp получает значение tmp=10/2=5. Далее по оператору **break** происходит выход из оператора **switch**. Если бы этого **break** не было, то еще бы выполнялся оператор помеченный **case 4**.

return (возврат)

Синтаксис

```
return (выражение);
```

Оператор return завершает выполнение функции, в которой он задан, и возвращает управление в вызывающую функцию, в точку, непосредственно следующую за вызовом.

Значение выражения, если оно задано, возвращается в вызывающую функцию в качестве значения вызываемой функции. Если выражение опущено, то возвращаемое значение не определено. Выражение может быть заключено в круглые скобки, хотя их наличие не обязательно.

Таким образом, использование оператора return необходимо либо для немедленного выхода из функции, либо для передачи возвращаемого значения.

Пример.

```
void main (void){
z = sum (5,1);
/* вызываем функцию sum и передаем ей значения 5 и 1,
результат вызова sum (5,1) отдаем z */
}

// функция суммирования
unsigned char sum (unsigned char x, unsigned char y)
{
return x+y;
}
```

Все перечисленные операторы довольно просты в логике своей работы. Однако их комбинации предоставляют программисту широкие возможности организации логики. Есть еще несколько операторов, но мы их рассмотрим по мере необходимости.

Избыточный займ и переполнение

Я пока не нашел подходящего места в самоучителе для этого вопроса, но считаю, что уже необходимо осветить этот вопрос. Что понимается под избыточным займом? Представим себе переменную tmp в которой сидит число 2. А теперь представим что из tmp вычитается число 3.

tmp = 2;
tmp = tmp – 3;

Сколько сейчас в tmp ? Во-первых, необходимо определиться с тем, к какому типу данных относитсятmp. Пусть это будет unsigned char (беззнаковый чар), т.е. его диапазон 0…255.

Во-вторых, для лучшего понимания от tmp будем отнимать в два захода, сначала отнимем 2, а потом 1.

Отняли 2. В tmp у нас 0.

Отнимаем еще 1. И значение tmp у нас становится 255. Произошел так называемый избыточный займ.

Необходимо понимать, что нет ограничений по поводу вычитания в беззнаковом чаре. Впрочем и в других переменных тоже. В знаковом чаре седьмой бит – это признак отрицательного числа. Но всё равно, если будет вычтено больше, чем есть в переменной, у данной переменной будет сделан избыточный займ. Т.е. число не станет более отрицательным, и оно не будет равно нулю.

Таким образом, механизм избыточного займа выглядит как вычитание из переменной до нуля, а затем вычитание остатка из максимума этой переменной (перечитайте абзац еще раз).

Переполнение выглядит точно также, но наоборот.

unsigned char tmp = 255;
tmp = tmp + 1;
Теперь в tmp у нас ноль.

Составление проекта из нескольких файлов исходников

Рано или поздно возникает ситуация, когда разработчик завершает один проект и приступает к написанию другого. Очевидно, что накопленные предыдущие знания (точнее говоря – куски программного кода) могут быть скопированы в новый проект. Однако, мы с вами ранее создавали всего лишь один файл (исходник на Си) в который всё сваливали в кучу и по мере желания пытались комментировать.

Сейчас мы с вами научимся создавать несколько файлов исходников которые будут работать в одном проекте. Зачем это надо? Несколько причин:

1) в разных файлах удобнее группировать функции по решению определенных задач (индикация, опрос клавиатуры, логика работы устройства, работа с EEPROM, управление другими устройствами, протоколыI2C, 1-Wire, RC5 и т.п.);
2) с каждым файлом визуальнее проще и нагляднее работать да и код не упирается «простынёй» в пол;
3) для будущих проектов элементарно подключить необходимые файлы и использовать готовые функции, что экономит время.

Итак, с помощью визарда мы создали проект. Затем создали рыбу

```
#include <pic.h>

__CONFIG (INTIO & UNPROTECT & LVPDIS & BOREN
```

```
& MCLRDIS & PWRTEN & WDTDIS);

void main (void)
{
}
```
и успешно откомпилировали.

Как создать новый файл исходника и подключить к проекту? Да точно также как и создать новую рыбу, только без текста (всё то же самое):

Нажимаем меню File – New. Появляется окно с именем Untitled. Нажимаем меню File – Save As… и приходим в путь с нашим проектом. Далее вводим имя файла (например, proba.c) и не забываем поставить галочку Add File To Project (Добавить Файл В Проект) + Сохранить. Всё.

А представьте ситуацию, когда вы уже имеете файл исходник с функциями по обслуживанию какого-то процесса, например, idikator.c. Здесь вам нужно вспомнить случай, когда вы забыли поставить галочку. Правой кнопкой мыши в окне дерева проекта щелкаем на ветке Source File (англ. – источники; сленг – сырцы) и выбираем пункт Add Files…, затем находим и открываем наш файл idikator.c . В дереве появляется наш файл. Это признак того, что файл подключен к проекту. Ура.

С чего начинается не «главный» исходник (где отсутствует функция main)? Рассмотрим конкретный пример. Пусть наш проект состоит из двух исходников. В главном исходнике мы конфигурируем МК, пишем main функцию из который инициализируем порты в функции podgot, затем входим в бесконечный цикл, который вызывает функцию мигалки из второго исходника.

Пишем в главный файл

```
#include <pic.h>

__CONFIG (INTIO & UNPROTECT & LVPDIS & BOREN
& MCLRDIS & PWRTEN & WDTDIS);

// === объявляем функции
void podgot (void); // подготовка МК
extern void migalka (void); // внешняя функция мигалки

// === главная функция
void main (void){ // начало функции
podgot (); // вызываем функцию подготовки
while(1)
{ // начало бесконечного цикла
migalka (); // внешняя функция мигалки
} // конец бесконечного цикла
} // конец функции

// === подготовка МК
void podgot (void){ // начало функции
INTCON = 0;  // чистим регистр прерываний
TRISA = 0b00000000; TRISB = 0b11111111; // направления
CMCON = 0x07; // отключение компараторов
PORTA = 0; PORTB = 0; // чистим порты
RBPU = 0; // подтягивающие R (1-откл, 0-вкл)
} // конец функции
```

пишем во второй файл

```
#include <pic.h>

// === сопоставление сигнальных линий
#define out1 RA6 // выход 1
#define out2 RA7 // выход 2

// === используемые функции
void pauza (void); // функция паузы
```

```
// === функция мигалки
void migalka (void){ // начало функции
out1 = 1; // установили ВЛУ
out2 = 0; // установили НЛУ
pauza (); // функция паузы
out1 = 1-out1; // инвертировать
out2 = 1-out2; // инвертировать
pauza (); // функция паузы
} // конец функции

// === функция паузы
void pauza (void){ // начало функции
unsigned int tmp; // локальная переменная
tmp = 0xffff; // в tmp поместить некое максимальное число
while (tmp-->0); /* выполнять декрементирование tmp
до тех пор, т.е. ПОКА tmp больше нуля */
}// конец функции
```

Компилим и смотрим в симуляторе. Работает.

По комментариям должно быть понятно, но мы акцентируем внимание. Для того, чтобы использовать функцию или глобальные переменные из одного файла исходника в другом файле исходнике необходимо представлять эти функции и переменные с помощью extern (внешний). Ну, например.

В одном файле у нас описано следующее:

```
void klava (void); // функция клавиатуры
bit knopka = 1; // инициализированный бит состояния кнопки
unsigned char rezim_n = 5; // инициализированный нумератор режима
unsigned char count; // переменная счетчика
```

Если в другом файле нам надо работать с этими функциями и переменными, то мы их обозначим (в другом файле) через extern

```
extern void klava (void); // внешняя функция клавиатуры
extern bit knopka; // внешний бит состояния кнопки
extern unsigned char rezim_n; // внешний нумератор режима
extern unsigned char count; // внешняя переменная счетчика
```

Что мы видим? Всё логично. Нельзя дважды инициализировать, т.е. присваивать некое значение переменным. Можно лишь использовать ранее инициализированное (или не инициализированное). С внешними переменными и функциями работаем также как и с обычными переменными, их только нужно представить через extern .

На что обратить внимание в примерах?

`TRISA = 0b00000000; TRISB = 0b11111111; // направления`

— бинарная запись нагляднее, сразу видно как работают линии.

```
out1 = 1-out1; // инвертировать
out2 = 1-out2; // инвертировать
```

— можно было тупо написать необходимое, но нужно знать способ инвертирования. Переменные типа`bit, char, int` и др. (bit не путать с линией) можно инвертировать как `tmp = ~tmp;` , ну или указанным способом `tmp = 1 - tmp;` .

unsigned int tmp; // локальная переменная
`tmp = 0xffff; while (tmp-->0);`

— простой способ организации задержки, где время будет составлять tmp*11 мкс (при частоте тактирования в 4 МГц), т.е. в нашем примере 65535*11 = 720885 мкс или 0,72 сек. Обращаю внимание, переменная tmp типа unsigned int (двухбайтное).

Массивы.

Массив – это хранилище переменных или констант (далее элементы массива). Мы говорим об элементах во множественном числе, соответственно, массив имеет некую размерность, т.е. указатель на количество элементов.

Рассмотрим несколько примеров.

`unsigned int array [3];`	массив array состоит из переменных типа unsigned int в количестве 3 шт.
`unsigned char arr_per [5] = {12, 23, 34, 45, 56};`	инициализированный массив arr_per состоит из переменных типа unsigned char в количестве 5 шт.
`const unsigned char arr_all [4] = {12, 0xEF, 0b00001111, 'A'};`	массив arr_all состоит из констант типа unsigned char в количестве 4 шт.
`extern unsigned char arr_per [5];`	внешний массив arr_per состоит из переменных типаunsigned char в количестве 5 шт.
`extern const unsigned char arr_all [4];`	внешний массив arr_all состоит из констант типа unsignedchar в количестве 4 шт.

Всё просто. Указываем тип элементов, затем некое имя и затем в квадратных скобках количество элементов. Если нам нужно инициализировать массив (т.е. определить содержимое), то ставим знак равно и в фигурных скобках через запятую перечисляем элементы.

Перечисление элементов в любом удобном для нас виде, как в этом примере const `unsigned char arr_all [4] = {12, 0xEF, 0b00001111, 'A'}`.

Указатель const с одной стороны говорит нам о том, что это константы (неизменяемые значения), а с другой стороны благодаря этому указателю массив располагается в памяти программ (во флеши). Массивы без указателя располагаются в оперативке (регистрах общего назначения).

В чём удобство массива? В нумерации элементов массива. Зачитывание элемента или изменение переменной в массиве будет происходить через её номер. **Нумерация элементов начинается с нуля**. Например

```
arr_all [4] = {12, 0xEF, 0b00001111, 'A'}
0й элемент = 12
1й элемент = 0xEF
2й элемент = 0b00001111
3й элемент = 'A' (символ 'A' = 0x41; см по таблице).
```

Следующие пример показывает, как прописать значения в массив

```
unsigned char arr_per [5] = {12, 23, 34, 45, 56};
// пропишем новые значения
arr_per [4] = 98;
arr_per [2] = 87;
arr_per [3] = 76;
arr_per [0] = 65;
arr_per [1] = 54;
```

Как видим, любой элемент массива можно изменить в любом месте и в любой последовательности. Но, как показывает практика, массив организуется с целью упорядочивания обращения к элементам, что и рассмотрим в следующем примере:

```
unsigned char arr_per [5] = {12, 23, 34, 45, 56};

void primer void{ // начало функции «пример»
unsigned char tmp; // некая локальная переменная tmp
for (tmp=0; tmp<4; tmp++) // цикл на основе tmp
// инкрементировать tmp пока tmp<4 и начать с нуля
PORTB = arr_per [tmp]; // в порт число из массива по номеру tmp
} // конец функции «пример»
```

Что мы прописываем в порт? Правильно – числа. А какие числа? Правильно – от нулевого до четвертого числа из массива. А какие нужны нам числа в массиве? Ну тут по ситуации в зависимости от того что подключено к порту. Рассмотрим пример подключения 1 разрядного семисегментного индикатора к порту.

```
/* электрические соединения
RB7 сегмент A
RB6 сегмент B
RB5 сегмент C
RB4 сегмент D
RB3 сегмент E
RB2 сегмент F
RB1 сегмент G
RB0 сегмент H
*/

// === массив констант с описанием 7-сегментных символов
const unsigned char arr_seg[12]={ // начало массива
// 0bABCDEFGH <- расположение сегментов по битам
 0b11111100, // 0й элемент, символ «0»
 0b01100000, // 1й элемент, символ «1»
 0b11011010, // 2й элемент, символ «2»
 0b11110010, // 3й элемент, символ «3»
 0b01100110, // 4й элемент, символ «4»
 0b10110110, // 5й элемент, символ «5»
 0b10111110, // 6й элемент, символ «6»
 0b11100000, // 7й элемент, символ «7»
 0b11111110, // 8й элемент, символ «8»
 0b11110110, // 9й элемент, символ «9»
 0b11000110, //10й элемент, символ градуса
 0b00000000  //11й элемент, пробел
}; // конец массива

// === функция отрисовки числа на порту
void ris_sim (unsigned char tmp) { // начало функции «отрисовка»
// в функцию передается параметр tmp
```

```
PORTB = arr_seg [tmp]; // прописать в порт число из массива по номеру tmp
} // конец функции «отрисовка»

// === используемые далее функции
void ris_sim (unsigned char tmp); // функция отрисовки числа на порту
void pauza (void); // функция паузы

// === функция перебора цифр
// (планируется вызывать из бесконечного цикла)
void perebor void{ // начало функции «перебор»
unsigned char tmp; // некая локальная переменная tmp
for (tmp=0; tmp<9; tmp++) // цикл на основе tmp
// инкрементировать tmp пока tmp<11 и начать с нуля
{ // начало цикла, т.к. в теле цикла несколько строк
ris_sim (tmp); // вызвать и передать в функцию ris_sim число tmp
pauza (); // функция паузы
} // конец цикла, т.к. в теле цикла несколько строк
} // конец функции «перебор»

// === функция паузы
void pauza (void){ // начало функции
unsigned int tmp; // локальная переменная
tmp = 0xffff; // в tmp поместить некое максимальное число
while (tmp-->0); /* выполнять декрементирование tmp
до тех пор, т.е. ПОКА tmp больше нуля */
}// конец функции
```

Смотрим. Мы имеем три функции

```
// === функция отрисовки числа на порту
// === функция перебора цифр (планируется вызывать из бесконечного цикла)
// === функция паузы
```

Запускаем из бесконечного цикла перебор цифр. В переборе запускается однократный цикл генерации числа tmp от 0 до 9. Это число передается в функцию отрисовки. Далее переданное число указывает, какой элемент взять из массива. В массиве у нас такие числа, которые при прописывании в порт формируют символы чисел от 0 до 9 (как видим, бинарные числа (битовые последовательности) в массиве гораздо нагляднее говорят нам о том, какие линии на порту будут включены). Для того чтобы перебор не был слишком быстрым, то мы вставляем задержку в виде функции паузы. Собственно и всё.

На что обратить внимание? Во всех функция фигурирует некая переменная tmp . Необходимо понимать, что в каждом конкретном месте это не одно и тоже. И об этом мы говорили, когда давали определение глобальных и локальных переменных. Нам так удобно называть однократно используемую локальную переменную. Можно сделать уникальное имя этой переменной в каждой взятой функции, но смысла напрягать мозг по именованию подобных переменных я не вижу.

Давайте упростим (сократим по содержанию) написанное выше, т.е. три функции объединим в одну.

```
// === функция перебора цифр (планируется вызывать из бесконечного цикла)
void perebor void{ // начало функции «перебор»
unsigned char tmp; // некая локальная переменная tmp
unsigned int tmp2; // локальная переменная tmp2
for (tmp=0; tmp<9; tmp++) // цикл на основе tmp
// инкрементировать tmp пока tmp<9 и начать с нуля
{ // начало цикла, т.к. в теле цикла несколько строк
PORTB = arr_seg [tmp]; // прописать в порт число из массива по номеру tmp
tmp2 = 0xffff; // в tmp2 поместить некое максимальное число
while (tmp2-->0); /* выполнять декрементирование tmp2
до тех пор, т.е. ПОКА tmp больше нуля */
} // конец цикла, т.к. в теле цикла несколько строк
} // конец функции «перебор»
```

Как-то так. Почему сразу не написали так? Вам нужно набивать руку в комбинировании функций и одновременно видеть и сравнивать одни с другими решениями. Выходной код уменьшится, но это не должно быть определяющим фактором в написании. Для вас важна прозрачность. Возможно, расписанный вариант из трех функций в каком-то месте окажется более полезным. Ну, например, если вам нужно использовать функцию задержки другими функциями. Или, например, если вам нужно на индикатор вывести какой-то определенный символ в любом месте программы, например «пятёрку», для этого достаточно в коде написать ris_sim (5); что означает нарисовать символ 5. Необходимо понимать, что кроме символов цифр, мы можем в массив прописать и символы некоторых букв, например, `A b Cd E F УРА` и даже пробел (увеличив размерность массива) J.

Не расслабляемся. Рассмотрим еще двухмерный массив.

`unsigned int array [5][6];`	массив array состоит из переменных типа unsigned int в котором 5 строк по 6 элементов.
`const unsigned char arr_all [2][3]={ {1,2,3}, {4,5,6} };`	массив констант arr_all состоит из переменных типа unsigned charв котором 2 строки по 3 элемента в каждой строке.

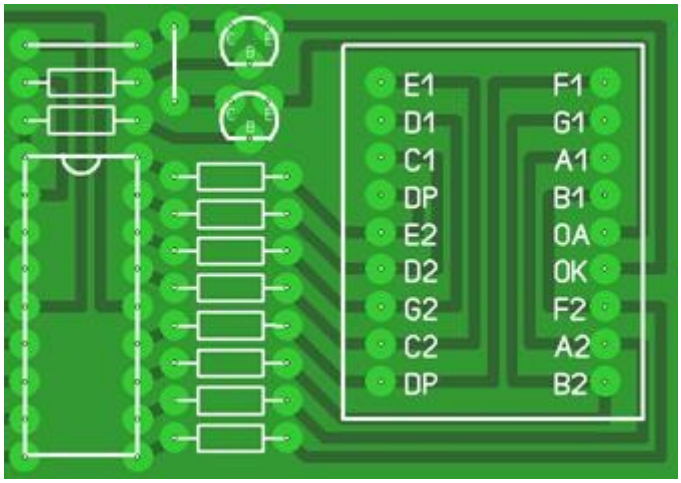
и другие варианты по аналогии с предыдущими.

Исключительно интересная вещь – двухмерный массив. Например, мы имеем графический индикатор 64*128 пикселей. Организуем соответствующий массив, в который можем поместить некий графический образ. Через простейшие циклы несложно организовать обмен с массивом (матрицей) вида array [Y][X] , где по YX будет число, определяющее включен или нет пиксель, а для цветных индикаторов возможен и цвет.

Приведем пример для начинающих. Представим что у нас два семисегментных индикатора. Представим, что у них разная распиновка. Усложним задачу – они

разного типа (один с общим анодом, другой с общим катодом). Можно два массива – для одного и для другого индикатора. А можно сделать компактно и красиво, с помощью двухмерного массива. Array [Y][X] – по Y будем различать первый и второй индикатор, а по X выводимые символы. Нам нужно будет только один раз напрячь мозг, чтобы прописать соответствующие битовые последовательности каждого символа для каждого разряда.

Вы можете возразить, что в практике вы такого не допустите. Ой, не зарекайтесь. Вот один из моих примеров печатной платы.



Это двухразрядный индикатор. Каждая линия управления объединяет разноименные сегменты (ну может где-то и одноименные). Удобно? Несомненно, односторонняя печатная плата с толстыми дорожками это всегда удобно (хоть маркером рисуй). Такой тип разводки индикатора я называю «греческим» рисунком, когда разноименные сегменты разных разрядов соединяются на плате одной дорожкой без перемычек.

На что обратить внимание? Посмотрите как в двухмерном массиве перечисляются элементы и как расставлены фигурные скобки. Строго говоря, можно перечислить в одну строчку.

Необходимо отметить, что массивы позволяют эффективно организовывать хранение и доступ к элементам с минимальными затратами программного кода. Так или иначе, в практике может возникнуть ситуация, когда объем массива глобальных переменных может физически не уместиться в текущем банке памяти (в нулевом банке). Для этого следует использовать префикс указывающий другой банк.

```
unsigned char bank1 arr_begushaya_stroka[180];
```

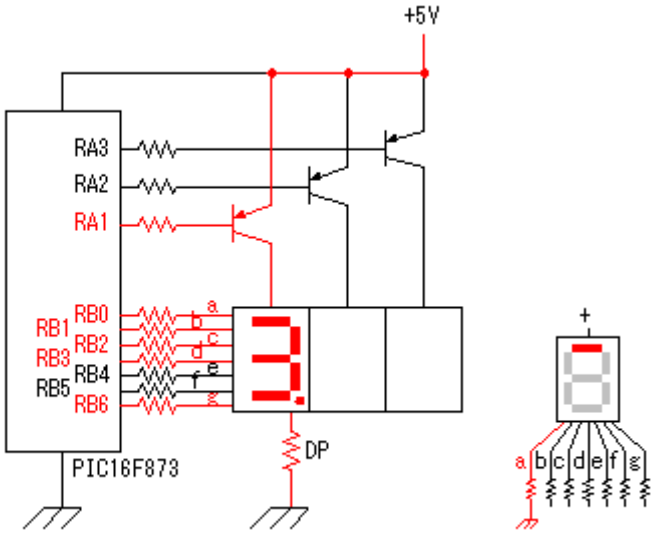
На этом краткое введение в массивы закончим. Знание двух основных видов и практическое их использование значительно облегчат ваш кодинг.

Динамическая индикация. Прерывания. Структуры

Бытует несколько типичных заблуждений по поводу 7-сегментных индикаторов и организации динамической индикации на них. Например, что для динамической индикации можно использовать только те индикаторы, где одноименные сегменты

объединены между собой. Или, например, сегменты индикатора следует подключать только к одному порту. Или, например, сегменты к порту должны подключаться по-порядку. Говоря словами классиков – believe me, всё гораздо проще и удобнее.

Рассмотрим схему включения трехразрядного индикатора (рисунок был готовый и он довольно наглядный по принципу работы).



Например, нам требуется отрисовать на индикаторе некое значение равное 3,16. Для этого мы последовательно зажигаем эти символы на каждом индикаторе с частотой комфортной для глаза 50*3=150 Гц. В результате картинка воспринимается нами, как одно целое. Для чего эта динамическая вообще нужна? У нас три индикатора, на каждом 8 сегментов (включая децимальную точку) и всего получается 3*8=24 сегмента. И для каждого сегмента требуется своя цифровая линия. А мы этого позволить не можем, т.к. в нашем учебном микроконтроллере не так уж и много ножек.

На что обратить внимание на схеме? На хитрое зажигание децимальной точки в первом разряде. Видимо точка нужна только в этом месте.