PicProg

A Library for Programming the PIC 12F629 via the Parallel Port

Dirk Bächle
dl9obn@darc.de

August 18, 2009

Contents

1 Introduction

PicProg is a small C library, providing a few functions to program a PIC 12F629 microcontroller via the parallel port. It is designed to work together with a special homemade programmer and the device driver "PPProg".
The following resources for the timings and programming specifications were used:

DS41190C *PIC12F629/675 Data Sheet, 8-Pin FLASH-Based 8-Bit CMOS Microcontrollers*

DS41191C *PIC12F629/75/PIC16F630/76 Memory Programming Specification*

DS31028A *In-Circuit Serial Programming Description*, being an excerpt from the

DS33023A *PICmicro(TM) Mid-Range MCU Family Reference Manual*

2 The library `picprog.c`

The basic structure of the library looks like this:

1a ⟨*picprog.c* 1a⟩≡
⟨*Header* 2b⟩
⟨*Include files* 2d⟩
⟨*Defines* 3a⟩
⟨*Global variables* 3b⟩
⟨*Functions* 3c⟩

2.1 The header file `picprog.h`

The according header file `picprog.h` has the following structure:

1b ⟨*picprog.h* 1b⟩≡
⟨*HF:Header* 2a⟩

```
#ifndef _PICPROG_H
#define _PICPROG_H
```

⟨*HF:Defines* 4a⟩
⟨*HF:Function prototypes* 14b⟩

```
#endif
```

Defines:
`_PICPROG_H`, never used.

2a ⟨*HF:Header* 2a⟩≡ (1b)
⟨*GPL disclaimer* 14c⟩

```
/** \file picprog.h
* Header file for the PIC 12F629 programming library and all programs using it.
```
⟨*Common disclaimer* 2c⟩

2.2 Header, includes and defines

Let us begin with the header of our library. Although NOWEB is already used to document the source, a lot of DOXYGEN commands are added, such that a short documentation for quick reference can be generated.

2b ⟨*Header* 2b⟩≡ (1a)
⟨*GPL disclaimer* 14c⟩

```
/** \file picprog.c
* Library for programming a PIC 12F629 via the parallel port under Linux.
```
⟨*Common disclaimer* 2c⟩

The common disclaimer as included to all source and header files:

2c ⟨*Common disclaimer* 2c⟩≡ (2)

```
* \author Dirk Baechle, dl9obn@darc.de
* \version 1.0
* \date 2005-10-29
*/

/* This file was created automatically from the file 'picprog.nw' by NOWEB.
* If you want to make changes, please edit the source 'picprog.nw'.
* A precompiled documentation can be found in 'picprog.pdf' and 'picprog.ps',
* respectively.
* Read it to understand why things are as they are. Thank you!
*/
```

Next are the include files, split into system header files and `picprog.h`.

2d ⟨*Include files* 2d⟩≡ (1a) 12a▷
⟨*Linux includes* 2e⟩

```
#include "ppprog.h"
#include "picprog.h"
```

2e ⟨*Linux includes* 2e⟩≡ (2d) 4c▷

```
#include <stdio.h>
```

2.3 Talking to the programmer

The programmer that is used, is attached to the parallel port and controlled by the following seven data lines (IN and OUT refer to the PIC programmer and not the PC!):

Line	Signal	Description
D0	Data In	
D1	Clock In	
D2	Data Enable	LOW=Data is sent to PIC HIGH=Data is read from PIC
D3	Vprog	LOW=Vprog is off HIGH=Vprog is on
D4	Vdd	LOW=Vdd is off HIGH=Vdd is on
D5	Clock Enable	LOW=Clock is enabled HIGH=Clock is blocked
ACK	Data Out	

We define them as constants...

3a ⟨*Defines* 3a⟩≡ (1a) 6a▷

```
static const int DATA_IN = 0x1;
static const int CLOCK_IN = 0x2;
static const int DATA_ENABLE = 0x4;
static const int VPROG = 0x08;
static const int VDD = 0x10;
static const int CLOCK_ENABLE = 0x20;
static const int DATA_OUT = 0x4000;
```

Defines:
`CLOCK_ENABLE`, never used.
`CLOCK_IN`, used in chunks 6c and 9b.
`DATA_ENABLE`, used in chunks 9 and 10a.
`DATA_IN`, never used.
`DATA_OUT`, used in chunk 9c.
`VDD`, used in chunks 5b, 6c, 9, and 10.
`VPROG`, used in chunks 5b, 6c, 9, and 10.

The device file is a global variable that gets initialized by a special function `initLib`.

3b ⟨*Global variables* 3b⟩≡ (1a)

```
/** The device file */
static int device_file = 0;
```

Defines:
`device_file`, used in chunks 3–7, 9, and 10.

2.3.1 Opening and closing the device file

3c ⟨*Functions* 3c⟩≡ (1a) 4b▷

```
/** Tries to open the device file and stores its handle
```

```
 * in the global variable \a device_file.
 * @return OK for success, ERROR for an error
 */
 int initLib()
 {
   /* Try to open device file */
   device_file = open("/dev/ppprog", O_RDONLY);
   if (device_file < 0)
   {
     device_file = 0;
     return ERROR;
   }

   return OK;
 }
```

Defines:
  `initLib`, used in chunks 14b and 16a.
Uses `device_file` 3b, `ERROR` 4a, and `OK` 4a.

4a ⟨*HF:Defines* 4a⟩≡ (1b)

```
#define ERROR 0
#define OK 1
```

Defines:
  `ERROR`, used in chunks 3c, 13b, 16a, 17c, 20, 22b, 24d, and 25a.
  `OK`, used in chunks 3c, 13d, and 21b.

4b ⟨*Functions* 3c⟩+≡ (1a) ◁3c 5a▷

```
/** Closes the device file.
*/
void closeLib()
{
  if (0 == device_file)
    return;

  close(device_file);
}
```

Defines:
  `closeLib`, used in chunks 14b, 15, 17, 20, 24, and 25.
Uses `device_file` 3b.

4c ⟨*Linux includes* 2e⟩+≡ (2d) ◁2e 23d▷

```
#include <fcntl.h>
#include <unistd.h>
#include <sys/ioctl.h>
```

2.3.2 Voltage control

Programming a PIC needs not only special timings for the data lines but for the applied voltages (Vprog and Vdd), too.
We define some functions for setting them ON and OFF, i.e. setting the programmer to reset state or program/verify mode.

5a ⟨*Functions* 3c⟩+≡ (1a) ◁4b 5b▷

```
/** Switches off all voltages.
*/
void setInactiveMode()
{
  int data[3];

  /* Switch everything off */
  data[0] = 1;
  data[1] = 0;
  data[2] = VDD_DELAY;

  /* Send command sequence */
  ioctl(device_file, IOCTL_PPPROG_SEND, data);

}
```

Defines:
`setInactiveMode`, used in chunks 14–17, 20, 21a, 24, and 25.
Uses `data` 25b, `device_file` 3b, and `VDD_DELAY` 6a.

5b ⟨*Functions* 3c⟩+≡ (1a) ◁5a 6b▷

```
/** Sets programmer to programming mode.
*/
void setProgrammingMode()
{
  int data[5];

  /* Switch on Vprog first */
  data[0] = 2;
  data[1] = VPROG;
  data[2] = VPROG_DELAY;
  /* and let Vdd follow */
  data[3] = VPROG | VDD;
  data[4] = VDD_DELAY;

  /* Send command sequence */
  ioctl(device_file, IOCTL_PPPROG_SEND, data);

}
```

Defines:
`setProgrammingMode`, used in chunks 14b, 16b, 17b, and 21a.

Uses data 25b, device_file 3b, VDD 3a, VDD_DELAY 6a, VPROG 3a, and VPROG_DELAY 6a.

The delay for VDD is rather large because the programming adapter uses a relay for switching...

6a ⟨*Defines* 3a⟩+≡ (1a) ◁3a 7a▷

```
/** Delay after raising Vprog */
static const int VPROG_DELAY = 100;
/** Delay after raising Vdd */
static const int VDD_DELAY = 100000;
```

Defines:
VDD_DELAY, used in chunk 5.
VPROG_DELAY, used in chunk 5b.

2.3.3 Basic read/write operations

A 'command word' tells the PIC what to do next. Incrementing its address pointer, loading a data word or programming an already loaded word to its memory. The six bits, that build the command, are sent with the LSB first.

6b ⟨*Functions* 3c⟩+≡ (1a) ◁5b 7b▷

```
void writeCommandWord(int command)
{
  int data[50];
  int i;

  data[0] = 12;

  for (i = 0; i < 6; ++i)
  {
    ⟨send a single bit 6c⟩

    command = command >> 1;
  }

  /* Send command sequence */
  ioctl(device_file, IOCTL_PPPROG_SEND, data);
}
```

Defines:
writeCommandWord, used in chunks 8 and 10–14.
Uses data 25b and device_file 3b.

For each single bit that is sent, the CLOCK line is raised, the bit is applied and gets accepted on the falling edge of the CLOCK to LOW.

6c ⟨*send a single bit* 6c⟩≡ (6b 7b)

```
/* Raise Clock and apply data */
data[i*4+1] = (command & 0x1) |
              CLOCK_IN |
```

```
                VPROG |
                VDD;
  /* Clock delay */
  data[i*4+2] = CLOCK_DELAY;

  /* Lower Clock while keeping data applied */
  data[i*4+3] = (command & 0x1) |
                VPROG |
                VDD;
  /* Clock delay */
  data[i*4+4] = CLOCK_DELAY;
```

Uses CLOCK_DELAY 7a, CLOCK_IN 3a, data 25b, VDD 3a, and VPROG 3a.

The different PIC commands are made available via defines...

7a ⟨*Defines* 3a⟩+≡ (1a) ◁6a 14a▷

```
  /** Clock delay */
  static const int CLOCK_DELAY = 100;

  /** Available command words */
  static const int LOAD_CONF = 0x00;
  static const int LOAD_DATA_PM = 0x02;
  static const int LOAD_DATA_DM = 0x03;
  static const int READ_DATA_PM = 0x04;
  static const int READ_DATA_DM = 0x05;
  static const int INCREMENT_ADDRESS = 0x06;
  static const int BEGIN_PROG = 0x08;
  static const int BEGIN_PROG_ET = 0x18;
  static const int END_PROG = 0x0A;
  static const int ERASE_PM = 0x09;
  static const int ERASE_DM = 0x0B;
```

Defines:
BEGIN_PROG, used in chunk 13a.
BEGIN_PROG_ET, never used.
CLOCK_DELAY, used in chunks 6c, 9, and 10.
END_PROG, never used.
ERASE_DM, used in chunk 11b.
ERASE_PM, used in chunk 11b.
INCREMENT_ADDRESS, used in chunk 11a.
LOAD_CONF, used in chunks 11b and 12c.
LOAD_DATA_DM, used in chunk 12b.
LOAD_DATA_PM, used in chunk 12b.
READ_DATA_DM, used in chunk 10c.
READ_DATA_PM, used in chunk 8.

'Writing a data word' means to send 16 bits to the PIC. Again the LSB is transferred first. By issueing a 'load data' command, followed by this routine and a 'program' command, data can be programmed to the memory of the PIC (see 2.3.7, p. 13).

7b ⟨*Functions* 3c⟩+≡ (1a) ◁6b 8▷

```
void writeDataWord(int command)
{
  int data[150];
  int i;

  data[0] = 32;

  command = (command & 0x3FFF) << 1;
  for (i = 0; i < 16; ++i)
  {
    ⟨send a single bit 6c⟩

    command = command >> 1;
  }

  /* Send command sequence */
  ioctl(device_file, IOCTL_PPPROG_SEND, data);
}
```

Defines:
`writeDataWord`, used in chunks 11, 12, and 14b.
Uses `data` 25b and `device_file` 3b.

For reading data words, the command has to be sent to the PIC first. Then—upon successive clock cycles—the single bits of the data appear on the line `ACK` (= `DATA_OUT`). Thus, for reading data its direction gets reversed by disabling `DATA_ENABLE`, i.e. setting it to `HIGH`.

8 ⟨*Functions* 3c⟩+≡ (1a) ◁7b 10c▷

```
int readDataWordPM()
{
  int data[5];
  int din[2];
  int word = 0;
  int mask = 0x1;
  int i;

  writeCommandWord(READ_DATA_PM);

  data[0] = 1;

  ⟨disable data line 9a⟩

  for (i = 0; i < 16; ++i)
  {
    ⟨raise clock 9b⟩
    ⟨read single bit 9c⟩
    ⟨lower clock 10a⟩
```

```
  }

  ⟨enable data line 10b⟩

  word = (word >> 1) & 0x3FFF;
  return word;
}
```

Defines:
  readDataWordPM, used in chunks 13d, 14b, and 16.
Uses data 25b, READ_DATA_PM 7a, word 25b, and writeCommandWord 6b.

9a ⟨*disable data line* 9a⟩≡ (8 10c)

```
/* Disable data line */
data[1] = DATA_ENABLE |
          VPROG |
          VDD;
/* Clock delay */
data[2] = CLOCK_DELAY;

/* Send command sequence */
ioctl(device_file, IOCTL_PPPROG_SEND, data);
```

Uses CLOCK_DELAY 7a, data 25b, DATA_ENABLE 3a, device_file 3b, VDD 3a, and VPROG 3a.

9b ⟨*raise clock* 9b⟩≡ (8 10c)

```
/* Raise Clock */
data[1] = CLOCK_IN |
          DATA_ENABLE |
          VPROG |
          VDD;
/* Clock delay */
data[2] = CLOCK_DELAY;

/* Send command sequence */
ioctl(device_file, IOCTL_PPPROG_SEND, data);
```

Uses CLOCK_DELAY 7a, CLOCK_IN 3a, data 25b, DATA_ENABLE 3a, device_file 3b, VDD 3a, and VPROG 3a.

9c ⟨*read single bit* 9c⟩≡ (8 10c)

```
/* Read data bit */
ioctl(device_file, IOCTL_PPPROG_CHECK, din);

if ((din[0] & DATA_OUT) == DATA_OUT)
  word |= mask;

mask = mask << 1;
```

Uses data 25b, DATA_OUT 3a, device_file 3b, and word 25b.

10a ⟨*lower clock* 10a⟩≡ (8 10c)

```
/* Lower Clock */
data[1] = DATA_ENABLE |
          VPROG |
          VDD;
/* Clock delay */
data[2] = CLOCK_DELAY;

/* Send command sequence */
ioctl(device_file, IOCTL_PPPROG_SEND, data);
```

Uses `CLOCK_DELAY` 7a, `data` 25b, `DATA_ENABLE` 3a, `device_file` 3b, `VDD` 3a, and `VPROG` 3a.

10b ⟨*enable data line* 10b⟩≡ (8 10c)

```
/* Enable data line */
data[1] = VPROG |
          VDD;
/* Clock delay */
data[2] = CLOCK_DELAY;

/* Send command sequence */
ioctl(device_file, IOCTL_PPPROG_SEND, data);
```

Uses `CLOCK_DELAY` 7a, `data` 25b, `device_file` 3b, `VDD` 3a, and `VPROG` 3a.

10c ⟨*Functions* 3c⟩+≡ (1a) ◁8 11a▷

```
int readDataWordDM()
{
  int data[5];
  int din[2];
  int word = 0;
  int mask = 0x1;
  int i;

  writeCommandWord(READ_DATA_DM);

  data[0] = 1;

  ⟨disable data line 9a⟩

  for (i = 0; i < 16; ++i)
  {
    ⟨raise clock 9b⟩
    ⟨read single bit 9c⟩
    ⟨lower clock 10a⟩
  }

  ⟨enable data line 10b⟩
```

```
  word = (word >> 1) & 0x3FFF;
  return word;
}
```

Defines:
  `readDataWordDM`, used in chunk 14b.
Uses `data` 25b, `READ_DATA_DM` 7a, `word` 25b, and `writeCommandWord` 6b.

Reading and writing does not automatically step the internal PIC address counter forward. For this a special 'command' exists, that has to be triggered to get to the next memory location.

11a ⟨*Functions* 3c⟩+≡ (1a) ◁10c 11b▷

```
void incrementAddress()
{
  writeCommandWord(INCREMENT_ADDRESS);
}
```

Defines:
  `incrementAddress`, used in chunks 11b, 14b, 16, 17c, 20, 24d, and 25a.
Uses `INCREMENT_ADDRESS` 7a and `writeCommandWord` 6b.

2.3.4 Bulk erase

For time delays of several ms, the functions `AlarmSet` and `AlarmWait` are used in the following... (see `alarm.c` and `alarm.h` in section 7, p. 26). The bulk erase procedure more or less follows the recommendations of the *PIC12F629 / 75 / PIC16F630 / 76 Memory Programming Specification* (DS41191C, p. 10).

11b ⟨*Functions* 3c⟩+≡ (1a) ◁11a 12b▷

```
void bulkErase()
{
  int i = 0;
  writeCommandWord(LOAD_CONF);
  writeDataWord(0x3FFF);
  for (; i < 7; ++i)
    incrementAddress();
  writeCommandWord(ERASE_PM);
  AlarmSet(9);
  AlarmWait();
  writeCommandWord(ERASE_DM);
  AlarmSet(9);
  AlarmWait();
}
```

Defines:
  `bulkErase`, used in chunks 14b, 17b, and 21a.
Uses `AlarmSet` 26, `AlarmWait` 26, `ERASE_DM` 7a, `ERASE_PM` 7a, `incrementAddress` 11a, `LOAD_CONF` 7a, `writeCommandWord` 6b, and `writeDataWord` 7b.

12a ⟨*Include files* 2d⟩+≡ (1a) ◁2d

```
#include "alarm.h"
```

2.3.5 Loading a data word

The 'load data' commands store the given data in the 'programming register' of the PIC. It can then be programmed to the memory of the PIC by issueing a 'program' command (see 2.3.7).

12b ⟨*Functions* 3c⟩+≡ (1a) ◁11b 12c▷

```
void loadDataWordPM(int data)
{
  writeCommandWord(LOAD_DATA_PM);
  writeDataWord(data);
}

void loadDataWordDM(int data)
{
  writeCommandWord(LOAD_DATA_DM);
  writeDataWord(data);
}
```

Defines:
  `loadDataWordDM`, used in chunk 14b.
  `loadDataWordPM`, used in chunks 13c and 14b.
Uses `data` 25b, `LOAD_DATA_DM` 7a, `LOAD_DATA_PM` 7a, `writeCommandWord` 6b, and `writeDataWord` 7b.

2.3.6 Loading config block

'Loading config block' means that the address pointer of the PIC is set to `0x2000`, the start of the memory area where the Config word resides...

12c ⟨*Functions* 3c⟩+≡ (1a) ◁12b 13a▷

```
void loadConfig()
{
  writeCommandWord(LOAD_CONF);
  writeDataWord(0x3FFF);
  AlarmSet(9);
  AlarmWait();
}
```

Defines:
  `loadConfig`, used in chunks 14b, 16d, and 20b.
Uses `AlarmSet` 26, `AlarmWait` 26, `LOAD_CONF` 7a, `writeCommandWord` 6b, and `writeDataWord` 7b.

2.3.7 Programming with verify

13a ⟨*Functions* 3c⟩+≡ (1a) ◁12c 13b▷

```
void programInternallyTimed()
{
  writeCommandWord(BEGIN_PROG);
  AlarmSet(150);
  AlarmWait();
}
```

Defines:
  `programInternallyTimed`, used in chunks 13c and 14b.
Uses `AlarmSet` 26, `AlarmWait` 26, `BEGIN_PROG` 7a, and `writeCommandWord` 6b.

Right after programming a data word–using the internally timed PIC operation—we verify it... if necessary up to `PROGRAM_MAX_TRY` times.

13b ⟨*Functions* 3c⟩+≡ (1a) ◁13a 21a▷

```
int programAndVerify(int data)
{
  int check = 0;
  int try = 0;
  for (; try < PROGRAM_MAX_TRY; ++try)
  {
    ⟨program data word 13c⟩
    ⟨verify data word 13d⟩
  }

  return ERROR;
}
```

Defines:
  `programAndVerify`, used in chunks 14b, 17c, 20, and 25a.
Uses `data` 25b, `ERROR` 4a, and `PROGRAM_MAX_TRY` 14a.

13c ⟨*program data word* 13c⟩≡ (13b)

```
loadDataWordPM(data);
programInternallyTimed();
```

Uses `data` 25b, `loadDataWordPM` 12b, and `programInternallyTimed` 13a.

13d ⟨*verify data word* 13d⟩≡ (13b)

```
check = readDataWordPM();
if (check == data)
{
  AlarmSet(150);
  AlarmWait();
  return OK;
}
```

Uses `AlarmSet` 26, `AlarmWait` 26, `data` 25b, `OK` 4a, and `readDataWordPM` 8.

14a ⟨*Defines* 3a⟩+≡ (1a) ◁7a 23b▷

```
#define PROGRAM_MAX_TRY 3
```

Defines:
PROGRAM_MAX_TRY, used in chunk 13b.

The following list contains nearly all functions that are offered to the user by our PicProg library.

14b ⟨*HF:Function prototypes* 14b⟩≡ (1b) 22a▷

```
extern int initLib();
extern void closeLib();
extern void setInactiveMode();
extern void setProgrammingMode();

extern void writeCommandWord(int command);
extern void writeDataWord(int data);

extern void incrementAddress();
extern int readDataWordPM();
extern int readDataWordDM();

extern void loadDataWordPM(int data);
extern void loadDataWordDM(int data);
extern void loadConfig();
extern void bulkErase();
extern void programInternallyTimed();
extern int programAndVerify(int data);
```

Uses bulkErase 11b, closeLib 4b, data 25b, incrementAddress 11a, initLib 3c, loadConfig 12c, loadDataWordDM 12b, loadDataWordPM 12b, programAndVerify 13b, programInternallyTimed 13a, readDataWordDM 10c, readDataWordPM 8, setInactiveMode 5a, setProgrammingMode 5b, writeCommandWord 6b, and writeDataWord 7b.

GPL disclaimer

The GNU GPL disclaimer, as used by all source files...

14c ⟨*GPL disclaimer* 14c⟩≡ (2)

```
/* PicProg - A library for programming the PIC 12F629 via the
*            parallel port under Linux.
* Copyright (C) 2005 by Dirk Baechle (dl9obn@darc.de)
*
* This program is free software; you can redistribute it and/or
* modify it under the terms of the GNU General Public License
* as published by the Free Software Foundation; either version 2
* of the License, or (at your option) any later version.
*
* This program is distributed in the hope that it will be useful,
* but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public
 * License along with this program; if not, write to the
 *
 * Free Software Foundation, Inc.
 * 675 Mass Ave
 * Cambridge
 * MA 02139
 * USA
 *
 */
```

3 First test program

The following program was used to test the basic functionality of the PPProg driver in connection with the homemade parallel port programming adapter. It does not reprogram the PIC but simply reads the `OSCCAL` value and the Config word, in order to ensure that communication between the adapter/PC and the PIC is established.

15 ⟨*testlib.c* 15⟩≡

```
#include <stdio.h>
#include "picprog.h"

int main(void)
{
  int picAddress = 0;
  int osccal = 0;
  int configword = 0;

  ⟨initialize library 16a⟩
  ⟨switch to programming mode 16b⟩
  ⟨read OSCCAL value 16c⟩
  printf("OSCCAL value 0x3FF is: %X\n", osccal);
  ⟨read CONFIG value 16d⟩
  printf("Config word is: %X\n", configword);

  setInactiveMode();
  closeLib();

  return 0;
}
```

Defines:
`main`, never used.

Uses closeLib 4b, configword 21b, osccal 21b, picAddress 21b, setInactiveMode 5a, and word 25b.

16a ⟨*initialize library* 16a⟩≡ (15 17a 25c)

```
if (ERROR == initLib())
{
  printf("Could not init PIC library!\n");
  return ERROR;
}
```

Uses ERROR 4a and initLib 3c.

16b ⟨*switch to programming mode* 16b⟩≡ (15 17a 21a)

```
setInactiveMode();
printf("Attach adapter to parallel port and press RETURN.\n");
getchar();
setProgrammingMode();
printf("Programming mode is ON now!\n");
printf("Press RETURN again to start.\n");
getchar();
```

Uses setInactiveMode 5a and setProgrammingMode 5b.

16c ⟨*read OSCCAL value* 16c⟩≡ (15 21a)

```
for (; picAddress < 0x3FF; ++picAddress)
  incrementAddress();

osccal = readDataWordPM();
```

Uses incrementAddress 11a, osccal 21b, picAddress 21b, and readDataWordPM 8.

16d ⟨*read CONFIG value* 16d⟩≡ (15 21a)

```
loadConfig();

incrementAddress();
incrementAddress();
incrementAddress();
incrementAddress();
incrementAddress();
incrementAddress();
incrementAddress();
configword = readDataWordPM();
```

Uses configword 21b, incrementAddress 11a, loadConfig 12c, and readDataWordPM 8.

4 Programming the PIC

After the test has been passed successfully, the PIC is about to be programmed the first time now. The basic structure of the program is similar to the previous test example:

17a ⟨*blinkprog.c* 17a⟩≡

```
#include <stdio.h>
#include "picprog.h"

⟨blink array 18⟩

int main(void)
{
  int picAddress = 0;
  int osccal = 0x349C;
  int configword = 0x3181;

  ⟨initialize library 16a⟩
  ⟨switch to programming mode 16b⟩
  ⟨bulk erase 17b⟩
  ⟨program blink array 17c⟩
  ⟨program OSCCAL value 20a⟩
  ⟨program CONFIG value 20b⟩
  setInactiveMode();
  closeLib();

  return 0;
}
```

Defines:
  `main`, never used.
Uses `closeLib` 4b, `configword` 21b, `osccal` 21b, `picAddress` 21b, and `setInactiveMode` 5a.

17b ⟨*bulk erase* 17b⟩≡ (17a)

```
printf("Bulk erasing memory...\n");
bulkErase();
printf("Done!\n");
setInactiveMode();
printf("Press RETURN to start programming.\n");
getchar();
setProgrammingMode();
```

Uses `bulkErase` 11b, `setInactiveMode` 5a, and `setProgrammingMode` 5b.

17c ⟨*program blink array* 17c⟩≡ (17a)

```
for (picAddress = 0; picAddress < BLINK_ARRAY_LEN; ++picAddress)
{
  if (ERROR == programAndVerify(blink_array[picAddress]))
```

```
  {
    printf("Error while programming address %X with data %X!\n", picAddress, blink_array
    setInactiveMode();
    closeLib();
    return ERROR;
  }
  incrementAddress();
}
```

Uses `blink_array` 18, `BLINK_ARRAY_LEN` 18, `closeLib` 4b, `data` 25b, `ERROR` 4a, `incrementAddress` 11a, `picAddress` 21b, `programAndVerify` 13b, and `setInactiveMode` 5a.

The test program consists of a simple infinite loop that toggles a port from `HIGH` to `LOW` and back all 500ms. It is defined as array of integers in our test program.

18 ⟨*blink array* 18⟩≡ (17a)

```
#define BLINK_ARRAY_LEN 29

static int blink_array[BLINK_ARRAY_LEN] =
{
0x2804,
0x0000,
0x0000,
0x0000,
0x1283,
0x0185,
0x3007,
0x0099,
0x1683,
0x303B,
0x0085,
0x1283,
0x3000,
0x0085,
0x2013,
0x3004,
0x0085,
0x2013,
0x280C,
0x30F9,
0x00A2,
0x30F9,
0x00A1,
0x0000,
0x0BA1,
0x2817,
0x0BA2,
0x2815,
0x0008
};
```

Defines:
`blink_array`, used in chunk 17c.
`BLINK_ARRAY_LEN`, used in chunk 17c.

Its assembler source looks something like this:

19 ⟨*blinkprog.asm* 19⟩≡

```
#include "p12f629.inc"

        __config 31A8

        org 0x0

        goto    0x04
        nop
        nop
        nop
        bcf     STATUS, RP0
        clrf    GPIO
        movlw   0x07
        movwf   CMCON
        bsf     STATUS, RP0
        movlw   0x3B
        movwf   0x05
        bcf     STATUS, RP0


loop:
        movlw   0x00
        movwf   GPIO
        call    delay
        movlw   0x04
        movwf   GPIO
        call    delay
        goto    loop

delay:
        movlw   0xF9
        movwf   0x22
outer:
        movlw   0xF9
        movwf   0x21
inner:
        nop
        decfsz  0x21,1
        goto inner

        decfsz  0x22,1
        goto    outer
```

```
        return

END
```

20a ⟨*program OSCCAL value* 20a⟩≡ (17a 21a)

```
for (; picAddress < 0x3FF; ++picAddress)
  incrementAddress();

if (ERROR == programAndVerify(osccal))
{
  printf("Error while programming OSCCAL value %X!\n", osccal);
  setInactiveMode();
  closeLib();
  return ERROR;
}
```

Uses `closeLib` 4b, `ERROR` 4a, `incrementAddress` 11a, `osccal` 21b, `picAddress` 21b, `programAndVerify` 13b, and `setInactiveMode` 5a.

20b ⟨*program CONFIG value* 20b⟩≡ (17a 21a)

```
loadConfig();

for ( picAddress = 0x2000; picAddress < 0x2007; ++picAddress)
  incrementAddress();

if (ERROR == programAndVerify(configword))
{
  printf("Error while programming CONFIG value %X!\n", configword);
  setInactiveMode();
  closeLib();
  return ERROR;
}
```

Uses `closeLib` 4b, `configword` 21b, `ERROR` 4a, `incrementAddress` 11a, `loadConfig` 12c, `picAddress` 21b, `programAndVerify` 13b, and `setInactiveMode` 5a.

5 Reading and programming INHEX files

Now we are able to program a PIC, basically, but changing the array `blink_array` in `blinkprog.c` is tedious—especially for larger programs. So what we need is a routine that reads an INHEX file, as produced by `gpasm` for example, and programs it to a PIC directly. There are several INHEX formats available that differ in the number of program words per line. In the following, we focus on INHEX16 only. Each line has the following format:

Index	Chars	Contents
0	1	The character ':'
1	2	Number n of program words
3	4	The address, the data should be written to
7	2	Zero byte '00': For padding to 2Byte boundary??
9	n*4	The n program words, 4Bytes each
9 + n*4	2	Checksum

However, there are two exceptions from this rule. First is the Config word line. A configuration word `0x31A8` would result in the line

```
:0120070031A8FF
```

The second 'special' line is the end line that looks like this:

```
:00000001FF
```

21a ⟨*Functions* 3c⟩+≡ (1a) ◁13b

```
int programInhex16File(const char *fileName)
{
  ⟨pihf: Variables 21b⟩

  ⟨open INHEX16 file 22b⟩

  ⟨switch to programming mode 16b⟩
  ⟨read OSCCAL value 16c⟩
  ⟨read CONFIG value 16d⟩
  bulkErase();
  setInactiveMode();
  setProgrammingMode();
  picAddress = 0;
  ⟨read and program lines 22d⟩
  ⟨program OSCCAL value 20a⟩
  ⟨program CONFIG value 20b⟩

  setInactiveMode();
  fclose(f);

  return status;
}
```

Defines:
`programInhex16File`, used in chunks 22a and 25c.
Uses `bulkErase` 11b, `f` 22c, `picAddress` 21b, `setInactiveMode` 5a, `setProgrammingMode` 5b, and `status` 21b.

21b ⟨*pihf: Variables* 21b⟩≡ (21a) 22c▷

```
int status = OK;
int picAddress = 0;
int osccal = 0;
int configword = 0;
```

Defines:
`configword`, used in chunks 15–17, 20b, and 23e.
`osccal`, used in chunks 15–17 and 20a.
`picAddress`, used in chunks 15–17, 20, 21a, 24d, and 25a.
`status`, used in chunk 21a.
Uses `OK` 4a.

22a ⟨*HF:Function prototypes* 14b⟩+≡ (1b) ◁14b

```
extern int programInhex16File(const char *fileName);
```

Uses `programInhex16File` 21a.

22b ⟨*open INHEX16 file* 22b⟩≡ (21a)

```
f = fopen(fileName, "r");
if (NULL == f)
{
  printf("Error: Could not open INHEX16 file %s for reading!\n", fileName);
  return ERROR;
}
```

Uses `ERROR` 4a and `f` 22c.

22c ⟨*pihf: Variables* 21b⟩+≡ (21a) ◁21b 23a▷

```
FILE *f;
```

Defines:
`f`, used in chunks 21 and 22.

Now we loop through the single lines of the input file and program all the data we can find...

22d ⟨*read and program lines* 22d⟩≡ (21a)

```
while (!feof(f))
{
  ⟨read next line 22e⟩
  ⟨check for last line 23c⟩
  ⟨handle Config word line 23e⟩
  ⟨skip empty lines 24a⟩
  ⟨program normal line 24b⟩
}
```

Uses `f` 22c.

22e ⟨*read next line* 22e⟩≡ (22d)

```
fgets(input, MAX_INPUT, f);
```

Uses `f` 22c, `input` 23a, and `MAX_INPUT` 23b.

23a ⟨*pihf: Variables* 21b⟩+≡ (21a) ◁22c 23f▷

```
char input[MAX_INPUT+2];
```

Defines:
`input`, used in chunks 22–25.
Uses `MAX_INPUT` 23b.

23b ⟨*Defines* 3a⟩+≡ (1a) ◁14a

```
/** Maximum length of an input line for INHEX16 files */
#define MAX_INPUT 1000
```

Defines:
`MAX_INPUT`, used in chunks 22e and 23a.
Uses `input` 23a.

If we find the last line, we stop programming the data...

23c ⟨*check for last line* 23c⟩≡ (22d)

```
if (0 == strncmp(input, ":00000001FF", 11))
  break;
```

Uses `input` 23a.

The used string functions require `string.h`.

23d ⟨*Linux includes* 2e⟩+≡ (2d) ◁4c

```
#include <string.h>
```

If the 'Config word line' is encountered, the new config word is extracted and stored for later when it gets written to its address `0x2007` in the 'Config memory' block. Only the lower three bytes get accepted because we have to preserve the bandgap calibration bits 12 and 13.

23e ⟨*handle Config word line* 23e⟩≡ (22d)

```
if (0 == strncmp(input, ":01200700", 9))
{
  /* Extract config bits, excluding bandgap cals */
  sscanf(input+9, "%XFF", &ivalue);
  configword = (ivalue & 0xFFF) | (configword & 0xF000);
  continue;
}
```

Uses `configword` 21b, `input` 23a, and `ivalue` 23f.

23f ⟨*pihf: Variables* 21b⟩+≡ (21a) ◁23a 25b▷

```
int ivalue = 0;
```

Defines:
`ivalue`, used in chunks 23–25.

Before actually processing and extracting the program data, we quickly check whether the line is long enough according to the number of words that are given in the first entry. At this point, empty lines get filtered out too.

24a ⟨*skip empty lines* 24a⟩≡ (22d)

```
/* Extract number of words to program */
sscanf(input+1,"%2X", &ivalue);
/* Sanity check: Is the line long enough? */
if (ivalue*4+11 > strlen(input))
  continue;
```

Uses `input` 23a and `ivalue` 23f.

24b ⟨*program normal line* 24b⟩≡ (22d)

⟨*extract start address for line* 24c⟩
⟨*increment address counter if necessary* 24d⟩
⟨*extract program words and write them* 25a⟩

24c ⟨*extract start address for line* 24c⟩≡ (24b)

```
sscanf(input+3, "%2X", &ivalue);
lineaddress = ivalue << 8;
sscanf(input+5, "%2X", &ivalue);
lineaddress += ivalue;
```

Uses `input` 23a, `ivalue` 23f, and `lineaddress` 25b.

It might happen that not all memory places are written consecutively. So we have to take care about keeping the PIC address counter in synch with the given start address of each line.

24d ⟨*increment address counter if necessary* 24d⟩≡ (24b)

```
/* Do a sanity check first */
if (lineaddress < picAddress)
{
  /* Stepping back is not allowed! */
  printf("Start address of line %X is below current PIC address %X! Aborting...\n", line

  setInactiveMode();
  closeLib();
  return ERROR;
}

/* Step PIC address forward, if necessary */
while (lineaddress > picAddress)
{
  incrementAddress();
  ++picAddress;
}
```

Uses closeLib 4b, ERROR 4a, incrementAddress 11a, lineaddress 25b, picAddress 21b, and setInactiveMode 5a.

An additional char pointer is used to step through the input line and extract the single program words...

25a ⟨*extract program words and write them* 25a⟩≡ (24b)

```
/* Extract number of words to program */
sscanf(input+1,"%2X", &ivalue);

i_ptr = input+9;
for (word = 0; word < ivalue; ++word)
{
  /* Extract data word */
  sscanf(i_ptr,"%4X", &data);
  /* Program data word */
  if (ERROR == programAndVerify(data))
  {
    printf("Error while programming memory location %04X: Writing %04X\n", picAddress, d
    printf("Input line is <%s>\n", input);
    printf("Aborting INHEX16 programming...\n");
    setInactiveMode();
    closeLib();
    return ERROR;
  }
  /* Increment address */
  incrementAddress();
  ++picAddress;

  /* Step to next word */
  i_ptr += 4;
}
```

Uses closeLib 4b, data 25b, ERROR 4a, i_ptr 25b, incrementAddress 11a, input 23a, ivalue 23f, picAddress 21b, programAndVerify 13b, setInactiveMode 5a, and word 25b.

25b ⟨*pihf: Variables* 21b⟩+≡ (21a) ◁23f

```
char *i_ptr;
int word = 0;
int lineaddress = 0;
int data = 0;
```

Defines:
data, used in chunks 5–10, 12–14, 17c, and 25a.
i_ptr, used in chunk 25a.
lineaddress, used in chunk 24.
word, used in chunks 8–10, 15, and 25a.

6 Test programming an INHEX16 file

Now, the following simple program is all we need to write the hex code `blinkprog.hex` to our PIC:

25c ⟨*progin16.c* 25c⟩≡

```
#include <stdio.h>
#include "picprog.h"

int main(void)
{
  ⟨initialize library 16a⟩

  programInhex16File("blinkprog.hex");

  setInactiveMode();
  closeLib();

  return 0;
}
```

Defines:
`main`, never used.
Uses `closeLib` 4b, `programInhex16File` 21a, and `setInactiveMode` 5a.

7 The alarm functions

26 ⟨*alarm.c* 26⟩≡

```
/* alarm.c -- seligman 6/92 */

/*
-- Implementation of alarm.h
*/

#include <stdio.h>
#include <signal.h>
#include <sys/time.h>

static int alarmPending = 0;  /* Nonzero when the alarm is set. */

static void ualarm();
static void AlarmHandler();

/* Default BSDSIGS for backwards compatibility of makefile */
#ifndef BSDSIGS
#ifndef POSIXSIGS
#ifndef OTHERSIGS
#define BSDSIGS
#endif
#endif
#endif
```

```
void AlarmSet(time)
    int time;
{
    alarmPending = 1;
#ifdef BSDSIGS
    signal(SIGALRM, AlarmHandler);
#else
#ifdef POSIXSIGS
    sigset(SIGALRM, AlarmHandler);
#else
    /* You're on your own... */
# endif /*POSIXSIGS*/
#endif /*BSDSIGS*/
    ualarm(1000 * time, 0);
}


/*
-- If an alarm signal is lurking (due to a prior call to SetAlarm), then
-- pause until it arrives.  This procedure could have simply been written:
--   if (alarmPending) pause();
-- but that allows a potential race condition.
*/
void AlarmWait()
{
#ifdef BSDSIGS
    long savemask = sigblock(sigmask(SIGALRM));
    if (alarmPending)
        sigpause(savemask);
    sigsetmask(savemask);
#else
#ifdef POSIXSIGS
    sighold(SIGALRM);
    if (alarmPending)
      sigpause(SIGALRM);
    sigrelse(SIGALRM);
#else
    /* You're on your own */
#endif /*POSIXSIGS*/
#endif /*BSDSIGS*/

}


static void ualarm(us)
    unsigned us;
{
    struct itimerval rttimer, old_rttimer;
```

```
    rttimer.it_value.tv_sec  = us / 1000000;
    rttimer.it_value.tv_usec = us % 1000000;
    rttimer.it_interval.tv_sec  = 0;
    rttimer.it_interval.tv_usec = 0;
    if (setitimer(ITIMER_REAL, &rttimer, &old_rttimer)) {
        perror("ualarm");
        exit(1);
    }
}


#ifdef __hpux

static void AlarmHandler(sig, code, scp)
    int sig;
    int code;
    struct sigcontext *scp;
{
    alarmPending = 0;
    /* Prevent alarm signal from interrupting any pending read. */
    if (scp->sc_syscall == SYS_READ)
        scp->sc_syscall_action = SIG_RESTART;
}

#else

static void AlarmHandler()
{
    alarmPending = 0;
}

#endif __hpux
```

Defines:
`AlarmHandler`, never used.
`alarmPending`, never used.
`AlarmSet`, used in chunks 11–13 and 28.
`AlarmWait`, used in chunks 11–13 and 28.
`BSDSIGS`, never used.
`ualarm`, never used.

28 ⟨*alarm.h* 28⟩≡

```
/* alarm.h -- seligman 6/92 */

/*
-- Routines for using the system interval timer to time beeps.  Useful
-- for implementing the functions in beep.h on systems that don't provide
-- a more straightforward BeepWait() equivalent.
--
-- These routines use the ALRM signal.
```

```
*/


/*
-- Set the alarm for a time specified in ms.
*/
void AlarmSet(/*int time*/);

/*
-- Wait for the alarm, or return immediately if the alarm isn't set.
*/
void AlarmWait();
```

Uses `AlarmSet` 26 and `AlarmWait` 26.

List of code chunks

This list was generated automatically by NOWEB. The numeral is that of the first definition of the chunk.

⟨*alarm.c* 26⟩
⟨*alarm.h* 28⟩
⟨*blink array* 18⟩
⟨*blinkprog.asm* 19⟩
⟨*blinkprog.c* 17a⟩
⟨*bulk erase* 17b⟩
⟨*check for last line* 23c⟩
⟨*Common disclaimer* 2c⟩
⟨*Defines* 3a⟩
⟨*disable data line* 9a⟩
⟨*enable data line* 10b⟩
⟨*extract program words and write them* 25a⟩
⟨*extract start address for line* 24c⟩
⟨*Functions* 3c⟩
⟨*Global variables* 3b⟩
⟨*GPL disclaimer* 14c⟩
⟨*handle Config word line* 23e⟩
⟨*Header* 2b⟩
⟨*HF:Defines* 4a⟩
⟨*HF:Function prototypes* 14b⟩
⟨*HF:Header* 2a⟩
⟨*Include files* 2d⟩
⟨*increment address counter if necessary* 24d⟩
⟨*initialize library* 16a⟩
⟨*Linux includes* 2e⟩
⟨*lower clock* 10a⟩
⟨*open INHEX16 file* 22b⟩
⟨*picprog.c* 1a⟩

⟨*picprog.h* 1b⟩
⟨*pihf: Variables* 21b⟩
⟨*progin16.c* 25c⟩
⟨*program blink array* 17c⟩
⟨*program CONFIG value* 20b⟩
⟨*program data word* 13c⟩
⟨*program normal line* 24b⟩
⟨*program OSCCAL value* 20a⟩
⟨*raise clock* 9b⟩
⟨*read and program lines* 22d⟩
⟨*read CONFIG value* 16d⟩
⟨*read next line* 22e⟩
⟨*read OSCCAL value* 16c⟩
⟨*read single bit* 9c⟩
⟨*send a single bit* 6c⟩
⟨*skip empty lines* 24a⟩
⟨*switch to programming mode* 16b⟩
⟨*testlib.c* 15⟩
⟨*verify data word* 13d⟩

Index

This is a list of identifiers used, and where they appear. Underlined entries indicate the place of definition.