Tyndall USB Programmer

2010-01-27

Generated by Doxygen 1.6.1

Wed Jan 27 18:18:40 2010

Contents

1 Main Page

Instructions for using the Tyndall USB Programmer

Installation

The executable file *tynprog.exe* should be placed in a directory that is in the PATH system variable. An easy way to do this is to copy it to the *utils/bin* directory of WinAVR.

The latest FTDI drivers should be installed. These can be found here: `http://www.ftdichip.com/FTDrivers.htm`. Version 2.06.00 is known to work, but other versions should be ok too.

If there are problems with *tynprog.exe*, tynprog_oss.exe can be used instead. This version does not require the FTDI drivers, but requires the open source software *libusb* filter driver to be installed: `http://libusb-win32.sourceforge.net/#installation` In order to use this version from within Programmer's Notepad, it should be renamed to *tynprog.exe*, replacing the other version.

The EEPROM in the FT232R chip must be programmed before *tynprog.exe* can recognise the board. This should be done when the board is first assembled, and will only need to be done once. The file *Tyndall25mmProg.xml* (for 25mm USB boards) or *Tyndall10mmProg.xml* must be programmed using FTProg: `http://www.ftdichip.com/Resources/Utilities.htm`

Usage

Connect the Tyndall USB programming board, with the transceiver layer that you want to program, to the PC/laptop. Note that it is not possible to program the 25mm Nordic nRF2401 board, or the 25mm ZigBee revA board using USB. This is because the SPI clock pin is not available on the connector.

Make sure the power switch on the programming board is in the correct position.

tynprog.exe can be called from Programmer's Notepad using the [WinAVR] Program command in the tools menu. To do this the `PROG_PORT` variable in *apps/makerules* should be set to `USB`, e.g.:

```
PROG_PORT = USB
```

tynprog.exe can also be called from the command line. Some examples for setting fuses, writing program code to Flash memory, and reading back the EEPROM:

```
tynprog.exe -m atmega_fuses -w FF91EF

tynprog.exe -w atmega_flash -w adcToRf.bin

tynprog.exe -m atmega_eeprom -r eeprom.bin

tynprog.exe -m nrf9e5 -i -w app.bin

tynprog.exe -m nrf9e5 -e
```

tynprog.exe works with only with raw binary files. Some compilers may only generate intel-hex formatted files. The tool *srec_cat* that is installed with WinAVR can be used to convert between the formats.

To convert from hex to binary (replace file1.∗ with real filenames):

```
srec_cat file1.hex --intel -o file1.bin -binary
```

To convert from binary to hex (replace file1.∗ with real filenames):

```
srec_cat file1.bin --binary -o file1.hex -intel
```

If a 25mm board is being programmed for the first time, the `'-s'` option must be used, as the default clock frequency is 1MHz which is not fast enough for full speed communication.

A header is always added when programming a 10mm board. This header is necessary for the nRF9E5 to be able to read the memory correctly. A file which contains the actual content that is programmed to the board is output called *nrf9e5_out.bin*.

Note that only one device can be programmed at a time, and therefore only one programming board should be connected before programming.

Command-line Options

```
Usage: tynprog [options]

Options:
  -c <frequency>     Value in MHz for crystal oscillator frequency. This is
                     only for the nRF9E5. Valid values are 4, 8, 12, 16 or 20.
                     If this option is not used, 20 will be assumed.

  -e                 Erases memory.

  -h                 Print this help message.

  -i                 To allow support for in-system reprogramming.

  -m <memory type>   Select memory that will be written to. Valid options are:
                     nrf9e5, atmega_flash, atmega_eeprom, or atmega_fuses.

  -r <file>          Read memory and store it in <file> (in binary format).
                     Overwrites any existing file of the same name.

  -s                 Use very slow clock speed for programming (9600 bits/s).

  -w <file>          Write <file> to memory with verification.

Only one operation (erase, read, or write) can be performed in a single
execution.

For fuses, replace <file> with <ext_byte><high_byte><low_byte>. Example fuse
settings to use the internal 8MHz clock are FF91E4 for Atmega128, and FF91E2
for ATmega1281.

The format of the file type is raw binary. External tools should be used to
convert to/from other formats (e.g. Intel-Hex format).

Only one device can be programmed at a time.
```

Information for Developers

This software works by interfacing with the FT232R chip, which has eight pins that can be used for bit-banging. This software communicates with the FT232R through the API provided by the D2XX device drivers. An array of bytes is passed to the FT232R and appears as parallel data on the eight pins, each of which can be set to be an input (high-impedance) or an output. The logic value of each pin can is also read back every time the pins are written to. This allows bi-directional communication.

In order to access the API, the files *ftd2xx.h* and *ftd2xx.lib* (available with the drivers) must be put in the same folder as the source files, and *ftd2xx.lib* must be linked to when compiling.

Alternatively, the FT232R can be accessed by the functions in libftdi. These functions are described and implemented in *ftdi.h* and *ftdi.c*, which are available from `http://www.intra2net.com/en/developer/libftdi/`. libftdi uses libusb to perform the actual interaction with the chip. This must be installed on the PC where the programmer will run. The functions in libusb are detailed in *usb.h*, and the library *libusb.a* must be linked to when compiling. The libusb driver, and other files are available from `http://libusb-win32.sourceforge.net/`

To build with the D2XX driver, `FTDI_USE_D2XX` must be defined when compiling, *ftd2xx.lib* must be linked to, and *ftd2xx.h* should be in the same folder as other files.

To build with libftdi and libusb, `FTDI_USE_LIBUSB` must be defined when compiling, *libusb.a* must be linked to, *ftdi.c* should be compiled and linked to, and *ftdi.h* and *usb.h* should be in the same folder as other files.

The CodeLite IDE (`http://codelite.org/`) has been successfully used to compile this, and a workspace file is included with the sources files (*tynprog.workspace*). This workspace contains two projects for linking with D2XX or libusb / libftdi.

The bit-banging mechanism is used to communicate with the target microcontroller using the SPI protocol. The functions for this are in *ft_spi.h*. Each microcontroller (AVR ATmegaXXX, and nRF9E5) uses different commands to control programming, so requires their own implementation of a programming interface. This is done in the *atmega.c* and *nrf9e5.c* files. The interface itself is defined by the `spiDeviceInfo_t` structure in *common.h*. Any new microcontrollers for which suppport is added should provide the same set of commands to allow them to be easily integrated, and *tynprog.c* shouldn't need to be modified.

There is a significant delay every time the FT232R is accessed. In order to speed-up the operation of the application, the number of calls to the functions in *ft_spi.h* should be reduced as much as possible. This may require buffering commands and sending them all at once.

Copyright and License

2 Data Structure Index

2.1 Data Structures

Here are the data structures with brief descriptions:

3 File Index

3.1 File List

Here is a list of all documented files with brief descriptions:

4 Data Structure Documentation

4.1 spiDeviceInfo_t Struct Reference

Structure for describing the SPI device with a description string and bitmasks for each of the pins.

```
#include <common.h>
```

Data Fields

- char description [64]

- uint8_t ncs
- uint8_t sck
- uint8_t miso
- uint8_t mosi
- uint32_t baudRate
- uint32_t memorySize
- uint32_t codeSize
- uint32_t writeAdds [4][2]
- void(∗ write)(const uint8_t ∗, uint32_t, uint32_t)
- void(∗ read)(uint8_t ∗, uint32_t, uint32_t)
- void(∗ erase)(void)
- void(∗ addHeader)(uint8_t ∗, uint32_t, bool)

4.1.1 Detailed Description

Structure for describing the SPI device with a description string and bitmasks for each of the pins.

4.1.2 Field Documentation

4.1.2.1 void(∗ spiDeviceInfo_t::addHeader)(uint8_t ∗, uint32_t, bool)

Function adds header info to code (data, size, isSupportReprog).

4.1.2.2 uint32_t spiDeviceInfo_t::baudRate

Baud rate of communications.

4.1.2.3 uint32_t spiDeviceInfo_t::codeSize

Size of memory that can be used for code.

4.1.2.4 char spiDeviceInfo_t::description[64]

Description string.

4.1.2.5 void(∗ spiDeviceInfo_t::erase)(void)

Function erases all data in the memory.

4.1.2.6 uint32_t spiDeviceInfo_t::memorySize

Total size of memory, in bytes.

4.1.2.7 uint8_t spiDeviceInfo_t::miso

data: Master In, Slave Out.

4.1.2.8 uint8_t spiDeviceInfo_t::mosi

data: Master Out, Slave In.

4.1.2.9 uint8_t spiDeviceInfo_t::ncs

Active low Chip Select.

4.1.2.10 void(∗ spiDeviceInfo_t::read)(uint8_t ∗, uint32_t, uint32_t)

Function reads bytes from the memory (data, address, size).

4.1.2.11 uint8_t spiDeviceInfo_t::sck

Serial Clock.

4.1.2.12 void(∗ spiDeviceInfo_t::write)(const uint8_t ∗, uint32_t, uint32_t)

Function programs bytes to the memory memory (data, address, size).

4.1.2.13 uint32_t spiDeviceInfo_t::writeAdds[4][2]

When supporting reprogramming, there will be large gaps in the code. This array stores addresses, where data should be written. There can be 4 different writes, and the start address, and write count is stored.

The documentation for this struct was generated from the following file:

- common.h

5 File Documentation

5.1 atmega.h File Reference

Header file for reading and writing to internal memory on ATmega microcontrollers.

Functions

- spiDeviceInfo_t ∗ atmega_init (memType_t memory, bool isSlowClock)
- void atmega_write (const uint8_t ∗data, uint32_t address, uint32_t size)
- void atmega_read (uint8_t ∗data, uint32_t address, uint32_t size)
- void atmega_erase (void)
- void atmega_addHeader (uint8_t ∗data, uint32_t rawCodeSize, bool isSupportReprog)
- void atmega_reset (void)
- void atmega_parseFuseValues (const char ∗fuseString, uint8_t data[3])
- void atmega_printFuseInfo (uint8_t fuses[3])

5.1.1 Detailed Description

Header file for reading and writing to internal memory on ATmega microcontrollers. References:

- Atmel, ATmega128 Datasheet.
- Atmel, ATmega325 Datasheet.
- Atmel, App Note AVR910: In-System Programming.
- Tyndall 25mm USB programming board schematic.

Date:

24-Jan-2010

Author:

Seán Harte

5.1.2 Function Documentation

5.1.2.1 void atmega_addHeader (uint8_t ∗ *data*, uint32_t *rawCodeSize*, bool *isSupportReprog*)

When reprogramming support is needed, some data is added to the binary code image.

```
/-------------------\  0x20000 (128kB)
|  Bootloader Code  |
|-------------------|  0x1F800 (127kB)
|       empty       |
|-------------------|  N + 0x10000
|     Application   |
|        Code       |
|-------------------|  0x10000 (63 kB)
|       empty       |
|-------------------|  0x0F808
```

```
|     App CRC       |    0x0F806
|     App length    |    0x0F804
|     App CRC       |    0x0F802
|     App length    |
|-------------------|  0x0F800 (63 kB)
|       empty       |
|-------------------|  N
|     Application   |
|        Code       |
\-------------------/  0x00000 (0 kB)
```

Parameters:

↔ ***data*** program data, must be 128kB is size.

rawCodeSize size of code being sent to function.

isSupportReprog if this is zero, nothing is done.

5.1.2.2 void atmega_erase (void)

Writes 0xFF to every location in the memory.

5.1.2.3 spiDeviceInfo_t∗ atmega_init (memType_t *memory*, bool *isSlowClock*)

Used to get details of the atmega chip that is present, and initialise SPI connection.

Parameters:

memory defines what type of memory will be programmed (Flash, EEPROM, or fuses).

isSlowClock is set when a slow clock speed is needed.

Returns:

pointer to structure containing description of memory, SPI pins and description in internal FT232R EEPROM.

5.1.2.4 void atmega_parseFuseValues (const char ∗*fuseString*, uint8_t *data*[3])

Converts Fuse string into 3byte array. Fuse should be 6 hexadecimal digits: `"<ext_byte><high_-byte><low_byte>"`

Parameters:

fuseString string of fuse values.

→ ***data*** bytes that will contain fuse values.

5.1.2.5 void atmega_printFuseInfo (uint8_t *fuses*[3])

Prints out information of what the fuse values mean. Only for ATmega128!

Parameters:

fuses Fuse value as 3 bytes: `"<ext_byte><high_byte><low_byte>"`

5.1.2.6 void atmega_read (uint8_t $*$ *data*, uint32_t *address*, uint32_t *size*)

Reads a number of bytes from memory. Flash, EEPROM, or fuses has been selected by atmega_init().

Parameters:

→ *data* where to store data. Must be big enough for memory type.

address where to start reading.

size how many bytes to read.

5.1.2.7 void atmega_reset (void)

Resets ATmega.

5.1.2.8 void atmega_write (const uint8_t $*$ *data*, uint32_t *address*, uint32_t *size*)

Programs a number of bytes to memory. Flash, EEPROM, or fuses has been selected by atmega_init().

Parameters:

data pointer to bytes that will be written to memory.

address where to start writing.

size How many bytes to write.

5.2 atmegaDevices.h File Reference

Where to add support for new ATmega devices.

Variables

- uint32_t atmega_infoTable [ ][5]
- const char $*$ atmega_strings [ ]

5.2.1 Detailed Description

Where to add support for new ATmega devices. This header makes it easy to add support for programming new ATmega devices. To add a new device a new row should be added to `atmega_infoTable` and `atmega_strings`. When the project is recompiled, the new device will be recognised.

References:

- Atmel, ATmega128 Datasheet.
- Atmel, ATmega1281 Datasheet.
- Atmel, ATmega325P Datasheet.
- Atmel, ATmega2561 Datasheet.

Date:

24-Jan-2010

Author:

Seán Harte

5.2.2 Variable Documentation

5.2.2.1 uint32_t atmega_infoTable[ ][5]

Initial value:

```
 {
    { ATMEGA_ATMEGA128_SIG, 131072,        256,    4096,          8},
    {ATMEGA_ATMEGA1281_SIG, 131072,        256,    4096,          8},
    {ATMEGA_ATMEGA2561_SIG, 262144,        256,    4096,          8},
    {ATMEGA_ATMEGA325P_SIG,  32768,        128,    1024,          4}
}
```

A number of different ATmega microcontrollers can be supported, as long as they support SPI memory programming. For example ATmega128, ATmega325, or ATmega2561. Note that this program uses page programming for Flash and slower byte programming for EEPROM as this is supported by most Atmegas. The ATmega is identified by it's signature bytes. The values in the table below show the details about memory sizes for each signature ID.

```
           Signature      | Flash |    Flash    | EEPROM |   EEPROM
             Bytes        | Size  | Page Size |  Size  | Page Size
```

5.3 common.h File Reference

Some defintions that can be used by all files. `#include <stdint.h>`

`#include <stdbool.h>`

`#include <time.h>`

Data Structures

- struct spiDeviceInfo_t

  Structure for describing the SPI device with a description string and bitmasks for each of the pins.

Defines

- #define MAX_MEMORY_SIZE 1048576
- #define SLOW_BAUDRATE 9600
- #define CMD_WINDOW_WIDTH 80
- #define MAX_VERIFY_ERRORS 10
- #define DESC_25MM_PROG "Tyndall 25mm USB Programmer"
- #define DESC_10MM_PROG "Tyndall 10mm Interface"
- #define LOC __FILE__":"STRING(__LINE__)
- #define BV(bit) (1 << (bit))

Enumerations

- enum memType_t

5.3.1 Detailed Description

Some defintions that can be used by all files.

Date:

27-Jan-2010

Author:

Seán Harte

5.3.2 Define Documentation

5.3.2.1 #define BV(bit) (1 << (bit))

Returns bit vector e.g. `BV(2)` is 00000010.

Parameters:

bit number of places to shift bit left.

5.3.2.2 #define CMD_WINDOW_WIDTH 80

Num of characters in one row of command window, (for progress bar).

5.3.2.3 #define DESC_10MM_PROG "Tyndall 10mm Interface"

String in FTDI EEPROM on 10mm programming boards.

5.3.2.4 #define DESC_25MM_PROG "Tyndall 25mm USB Programmer"

String in FTDI EEPROM on 25mm programming boards.

5.3.2.5 #define LOC __FILE__":"STRING(__LINE__)

Macro to build filename:linenumber string for debugging.

5.3.2.6 #define MAX_MEMORY_SIZE 1048576

Maximum size of any memory file (1MB).

5.3.2.7 #define MAX_VERIFY_ERRORS 10

Sets the maximum number of errors that will be printed by the verification process before quitting. 0 = infinite

5.3.3 Enumeration Type Documentation

5.3.3.1 enum memType_t

Different types of memory that can be programmed.

5.4 crc16.h File Reference

Interface for CRC functions.

Functions

- void crc16_init (void)
- uint16_t crc16_read (void)
- void crc16_update (uint8_t data)

5.4.1 Detailed Description

Interface for CRC functions. Function for calculating CRC-16 checksum based on the polynomial: x^16 + x^15 + x^2 + 1 (0xa001). CRC-16 is commonly used in disk-drive applications.

Example usage to calculate CRC of values in an array:

```
uint16_t crcValue;
crc16_init();
for (i = 0; i < SIZE; i++) {
    crc16_update(array[i]);
}
crcValue = crc16_read();
```

Date:

24-Jan-2010

Author:

Seán Harte

5.4.2 Function Documentation

5.4.2.1 void crc16_init (void)

Resets CRC value for a new calculation.

5.4.2.2 uint16_t crc16_read (void)

Read current CRC value.

Returns:

16-bit CRC value.

5.4.2.3 void crc16_update (uint8_t *data*)

Calculates new CRC value based on current value and new data byte.

Parameters:

data byte to use in updating CRC.

5.5 ftSpi.h File Reference

Header file for using FT232R to do SPI communications by bit-banging.

Defines

- #define FTSPI_MAX_BAUDRATE 921600

Functions

- void ftSpi_init (spiDeviceInfo_t ∗device)
- void ftSpi_readWrite (uint8_t ∗misoData, const uint8_t ∗mosiData, uint32_t count, bool chipSelect)
- void ftSpi_enableChipSelect (void)
- void ftSpi_disableChipSelect (void)
- void ftSpi_close (void)

5.5.1 Detailed Description

Header file for using FT232R to do SPI communications by bit-banging. Note that these functions will simply exit the application if there is an error communicating with the FT232R.

The calls to these functions take quite some time. To speed up the application try to call these functions as few times as possible (send as much data together as you can).

Date:

27-Jan-2010

Author:

Seán Harte

5.5.2 Define Documentation

5.5.2.1 #define FTSPI_MAX_BAUDRATE 921600

FTDI AN232R application note recommends less than 1Mbps. The actual clock rate is 16 times this value.

5.5.3 Function Documentation

5.5.3.1 void ftSpi_close (void)

Closes the FT232R device. Should be called before the program exits.

5.5.3.2 void ftSpi_disableChipSelect (void)

Brings chip select high. (Also, note that SCK will be low)

5.5.3.3 void ftSpi_enableChipSelect (void)

Brings chip select low. (Also, note that SCK will be low).

5.5.3.4 void ftSpi_init (spiDeviceInfo_t ∗ *device*)

Initialises FT232R device in bit-bang mode for SPI communication. This must called before using the other functions in this header.

Parameters:

device description of the SPI device that will be communicated with. This contains a string description which must match the string in in the FT232R internal EEPROM, and also defines which pins to use for MISO, MOSI, SCK, and nCS.

5.5.3.5 void ftSpi_readWrite (uint8_t ∗ *misoData*, const uint8_t ∗ *mosiData*, uint32_t *count*, bool *chipSelect*)

Simultaneously reads and writes a number of bytes using the SPI protocol by bit-banging over the FT232R. Note that each call to this function takes a relatively large time to return, so try to call it as few times as possible. The calling code should send as much data as it can with one call to this function.

Parameters:

→ ***misoData*** where to store bytes that are read. If this is NULL, then read data will be ignored.

mosiData pointer to bytes to write.

count how many bytes to read/write.

chipSelect if `true` then the chipSelect line is enabled before communicating, and disabled afterwards; otherwise the chipSelect line is not changed.

5.6 mainpage.h File Reference

Main documentation information.

5.6.1 Detailed Description

Main documentation information. Contains no code!

Date:

27-Jan-2010

Author:

Seán Harte

5.7 nrf9e5.h File Reference

Header file for reading and writing to the 25320 EEPROM used by the nRF9e5.

Functions

- spiDeviceInfo_t ∗ nrf_init (uint32_t freq)
- void nrf_write (const uint8_t ∗data, uint32_t address, uint32_t size)
- void nrf_read (uint8_t ∗data, uint32_t address, uint32_t size)
- void nrf_erase (void)
- void nrf_addHeader (uint8_t ∗data, uint32_t rawCodeSize, bool isSupportReprog)

5.7.1 Detailed Description

Header file for reading and writing to the 25320 EEPROM used by the nRF9e5. References:

- Atmel, AT25320 Datasheet.
- Nordic, nRF9e5 Datasheet.
- Tyndall 10mm nRF9E5 revA/B Schematic.
- Tyndall 10mm Interface Board revB/C Schematic.

Date:

24-Jan-2010

Author:

Seán Harte

5.7.2 Function Documentation

5.7.2.1 void nrf_addHeader (uint8_t ∗ *data*, uint32_t *rawCodeSize*, bool *isSupportReprog*)

Adds 3 byte header to EEPROM data that is required by nRF9E5. Also adds CRC and program size information which is needed for in-system programming. If in-system programming is wanted, then this function also copies the lower-half of the memory to the upper half.

```
byte[409-1]
   .
   .                = Unused
   .
byte[(N+1)*256]
byte[((N+1)*256)-1]
   .
   .                = Program data
   .
byte[7]
byte[6]             = High byte of program CRC
byte[5]             = Low byte of program CRC
byte[4]             = High byte of program size
byte[3]             = Low byte of program size
byte[2]             = How many 256 byte blocks are in the program (N)
byte[1]             = Where program starts (7)
byte[0]             = EEPROM configuration
```

Parameters:

↔ ***data*** program data.

rawCodeSize size of code being sent to function.

isSupportReprog if this is non-zero, then the data is copied to upper half of memory.

5.7.2.2 void nrf_erase (void)

Writes 0xFF to every location in the EEPROM.

5.7.2.3 spiDeviceInfo_t∗ nrf_init (uint32_t *freq*)

Used to get details of the 25320 EEPROM, and initialise connection.

Parameters:

freq frequency of crystal oscillator in Mhz (4, 8, 12, 16, or 20).

Returns:

Pointer to structure containing description of memory, SPI pins and description in internal FT232R EEPROM.

5.7.2.4 void nrf_read (uint8_t ∗ *data*, uint32_t *address*, uint32_t *size*)

Reads a number of bytes from the EEPROM. *address* + *size* must be <= 4093.

Parameters:

→ ***data*** location to store read data.

address where to start reading from.

size how many bytes to read.

5.7.2.5 void nrf_write (const uint8_t ∗ *data*, uint32_t *address*, uint32_t *size*)

Programs a number of bytes to the EEPROM. Header should be added before calling this function:

See also:

nrf_addEepromHeader(). *address* + *size* must be <= 4093.

Parameters:

data pointer to bytes to write to memory.

address where to start writing to.

size number of bytes to write to the memory, other bytes will be 0.

Index