/\*\*

\* @mainpage Pico Beacon

\*

\* @section overview\_sec Overview

\*

\* The Pico Beacon is an APRS based tracking beacon that operates in the UHF 420-450MHz band. The device utilizes a

\* Microchip PIC 18F2525 embedded controller, Motorola M12+ GPS engine, and Analog Devices AD9954 DDS. The device is capable

\* of generating a 1200bps A-FSK and 9600 bps FSK AX.25 compliant APRS (Automatic Position Reporting System) message.

\*

\* @section history\_sec Revision History

\*

\* @subsection v305 V3.05

\* **23 Dec 2006**, Change include; (1) change printf format width to conform to ANSI standard when new CCS 4.xx compiler released.

\*

\*

\* @subsection v304 V3.04

\* **10 Jan 2006**, Change include; (1) added amplitude control to engineering mode,

\* (2) corrected number of bytes reported in log,

\* (3) add engineering command to set high rate position reports (5 seconds), and

\* (4) corrected size of LOG\_COORD block when searching for end of log.

\*

\* @subsection v303 V3.03

\* **15 Sep 2005**, Change include; (1) removed AD9954 setting SDIO as input pin,

\* (2) additional comments and Doxygen tags,

\* (3) integration and test code calculates DDS FTW,

\* (4) swapped bus and reference analog input ports (hardware change),

\* (5) added message that indicates we are reading flash log and reports length,

\* (6) report bus voltage in 10mV steps, and

\* (7) change log type enumerated values to XORed nibbles for error detection.

\*

\*

\* @subsection v302 V3.02

\* **6 Apr 2005**, Change include; (1) corrected tracked satellite count in NMEA-0183 $GPGGA message,

\* (2) Doxygen documentation clean up and additions, and

\* (3) added integration and test code to baseline.

\*

\*

\* @subsection v301 V3.01

\* **13 Jan 2005**, Renamed project and files to Pico Beacon.

\*

\*

\* @subsection v300 V3.00

\* **15 Nov 2004**, Change include; (1) Micro Beacon extreme hardware changes including integral transmitter,

\* (2) PIC18F2525 processor,

\* (3) AD9954 DDS support functions,

\* (4) added comments and formatting for doxygen,

\* (5) process GPS data with native Motorola protocol,

\* (6) generate plain text $GPGGA and $GPRMC messages,

\* (7) power down GPS 5 hours after lock,

\* (8) added flight data recorder, and

\* (9) added diagnostics terminal mode.

\*

\*

\* @subsection v201 V2.01

\* **30 Jan 2004**, Change include; (1) General clean up of in-line documentation, and

\* (2) changed temperature resolution to 0.1 degrees F.

\*

\*

\* @subsection v200 V2.00

\* **26 Oct 2002**, Change include; (1) Micro Beacon II hardware changes including PIC18F252 processor,

\* (2) serial EEPROM,

\* (3) GPS power control,

\* (4) additional ADC input, and

\* (5) LM60 temperature sensor.

\*

\*

\* @subsection v101 V1.01

\* **5 Dec 2001**, Change include; (1) Changed startup message, and

\* (2) applied SEPARATE pragma to several methods for memory usage.

\*

\*

\* @subsection v100 V1.00

\* **25 Sep 2001**, Initial release. Flew ANSR-3 and ANSR-4.

\*

\*

\*

\* @section copyright\_sec Copyright

\*

\* Copyright (c) 2001-2009 Michael Gray, KD7LMO

\*

\*

\* @section gpl\_sec GNU General Public License

\*

\* 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., 59 Temple Place, Suite 330, Boston, MA 02111-1307 USA

\*

\*

\*

\* @section design Design Details

\*

\* Provides design details on a variety of the components that make up the Pico Beacon.

\*

\* @subpage power

\*/

/\*\*

\* @page power Power Consumption

\*

\* Measured DC power consumption.

\*

\* **3VDC prime power current**

\*

\* 7mA Held in reset

\* 18mA Processor running, all I/O off

\* 110mA GPS running

\* 120mA GPS running w/antenna

\* 250mA DDS running and GPS w/antenna

\* 420mA DDS running, GPS w/antenna, and PA chain on with no RF

\* 900mA Transmit

\*

\*/

// Hardware specific configuration.

#include <18f2525.h>

#device ADC=10

// NOTE: Even though we are using an external clock, we set the HS oscillator mode to

// make the PIC 18F252 work with our external clock which is a clipped 1V P-P sine wave.

#fuses HS,NOWDT,NOPROTECT,NOPUT,NOBROWNOUT,NOLVP

// C runtime library definitions.

#include

#include

// These compiler directives set the clock, SPI/I2C ports, and I/O configuration.

// TCXO frequency

#use delay(clock=19200000)

// Engineering and data extracation port.

#use rs232(baud=57600, xmit=PIN\_B7, rcv=PIN\_B6, STREAM=PC\_HOST)

// GPS engine

#use rs232(baud=9600, xmit=PIN\_C6, rcv=PIN\_C7)

#use i2c (master, scl=PIN\_C3, sda=PIN\_C4)

#use fast\_io(A)

#use fast\_io(B)

#use fast\_io(C)

// We define types that are used for all variables. These are declared

// because each processor has a different sizes for int and long.

// The PIC compiler defines int8\_t, int16\_t, and int32\_t.

/// Boolean value { false, true }

typedef boolean bool\_t;

/// Signed 8-bit number in the range -128 through 127.

typedef signed int8 int8\_t;

/// Unsigned 8-bit number in the range 0 through 255.

typedef unsigned int8 uint8\_t;

/// Signed 16-bit number in the range -32768 through 32767.

typedef signed int16 int16\_t;

/// Unsigned 16-bit number in the range 0 through 65535.

typedef unsigned int16 uint16\_t;

/// Signed 32-bit number in the range -2147483648 through 2147483647.

typedef signed int32 int32\_t;

/// Unsigned 32-bit number in the range 0 through 4294967296.

typedef unsigned int32 uint32\_t;

// Function and structure prototypes. These are declared at the start of

// the file much like a C++ header file.

// Map I/O pin names to hardware pins.

/// Heartbeat LED - Port A2

#define IO\_LED PIN\_A2

/// AD9954 DDS Profile Select 0 - Port A3

#define IO\_PS0 PIN\_A3

/// UHF amplifier and PA chain - Port A4

#define IO\_PTT PIN\_A4

/// AD9954 DDS Update - Port A5

#define IO\_UPDATE PIN\_A5

/// AD9954 CS (Chip Select) - Port B0

#define IO\_CS PIN\_B0

/// GPS Engine Power - Port B1

#define IO\_GPS\_PWR PIN\_B1

/// AD9954 DDS Profile Select 1 - Port C0

#define IO\_PS1 PIN\_C0

/// AD9954 DDS OSK (Output Shift Key) - Port C2

#define IO\_OSK PIN\_C2

/// GPS engine serial transmit pin - Port C6

#define IO\_GPS\_TXD PIN\_C6

// Public methods, constants, and data structures for each class.

/// Operational modes of the AD9954 DDS for the ddsSetMode function.

enum DDS\_MODE

{

/// Device has not been initialized.

DDS\_MODE\_NOT\_INITIALIZED,

/// Device in lowest power down mode.

DDS\_MODE\_POWERDOWN,

/// Generate FM modulated audio tones.

DDS\_MODE\_AFSK,

/// Generate true FSK tones.

DDS\_MODE\_FSK

};

void ddsInit();

void ddsSetAmplitude (uint8\_t amplitude);

void ddsSetOutputScale (uint16\_t amplitude);

void ddsSetFSKFreq (uint32\_t ftw0, uint32\_t ftw1);

void ddsSetFreq (uint32\_t freq);

void ddsSetFTW (uint32\_t ftw);

void ddsSetMode (DDS\_MODE mode);

void flashErase();

uint8\_t flashGetByte ();

void flashReadBlock(uint32\_t address, uint8\_t \*block, uint16\_t length);

void flashSendByte(uint8\_t value);

void flashSendAddress(uint32\_t address);

void flashWriteBlock(uint32\_t address, uint8\_t \*block, uint8\_t length);

/// Type of GPS fix.

enum GPS\_FIX\_TYPE

{

/// No GPS FIX

GPS\_NO\_FIX,

/// 2D (Latitude/Longitude) fix.

GPS\_2D\_FIX,

/// 3D (Latitude/Longitude/Altitude) fix.

GPS\_3D\_FIX

};

/// GPS Position information.

typedef struct

{

/// Flag that indicates the position information has been updated since it was last checked.

bool\_t updateFlag;

/// Month in UTC time.

uint8\_t month;

/// Day of month in UTC time.

uint8\_t day;

/// Hours in UTC time.

uint8\_t hours;

/// Minutes in UTC time.

uint8\_t minutes;

/// Seconds in UTC time.

uint8\_t seconds;

/// Year in UTC time.

uint16\_t year;

/// Latitude in milli arc-seconds where + is North, - is South.

int32\_t latitude;

/// Longitude in milli arc-seconds where + is East, - is West.

int32\_t longitude;

/// Altitude in cm

int32\_t altitudeCM;

/// Calculated altitude in feet

int32\_t altitudeFeet;

/// 3D speed in cm/second.

uint16\_t vSpeed;

/// 2D speed in cm/second.

uint16\_t hSpeed;

/// Heading units of 0.1 degrees.

uint16\_t heading;

/// DOP (Dilution of Precision)

uint16\_t dop;

/// 16-bit number that represents status of GPS engine.

uint16\_t status;

/// Number of tracked satellites used in the fix position.

uint8\_t trackedSats;

/// Number of visible satellites.

uint8\_t visibleSats;

} GPSPOSITION\_STRUCT;

void gpsInit();

bool\_t gpsIsReady();

GPS\_FIX\_TYPE gpsGetFixType();

int32\_t gpsGetPeakAltitude();

void gpsPowerOn();

bool\_t gpsSetup();

void gpsUpdate();

int16\_t lm92GetTemp();

/// Define the log record types.

enum LOG\_TYPE

{

/// Time stamp the log was started.

LOG\_BOOTED = 0xb4,

/// GPS coordinates.

LOG\_COORD = 0xa5,

/// Temperature

LOG\_TEMPERATURE = 0x96,

/// Bus voltage.

LOG\_VOLTAGE = 0x87

};

void logInit();

uint32\_t logGetAddress();

void logType (LOG\_TYPE type);

void logUint8 (uint8\_t value);

void logInt16 (int16\_t value);

bool\_t serialHasData();

void serialInit();

uint8\_t serialRead();

void serialUpdate();

uint16\_t sysCRC16(uint8\_t \*buffer, uint8\_t length, uint16\_t crc);

void sysInit();

void sysLogVoltage();

/// 0% duty cycle (LED Off) constant for function timeSetDutyCycle

#define TIME\_DUTYCYCLE\_0 0

/// 10% duty cycle constant for function timeSetDutyCycle

#define TIME\_DUTYCYCLE\_10 1

/// 70% duty cycle constant for function timeSetDutyCycle

#define TIME\_DUTYCYCLE\_70 7

uint8\_t timeGetTicks();

void timeInit();

void timeSetDutyCycle (uint8\_t dutyCycle);

void timeUpdate();

/// Operational modes of the TNC for the tncSetMode function.

enum TNC\_DATA\_MODE

{

/// No operation waiting for setup and configuration.

TNC\_MODE\_STANDBY,

/// 1200 bps using A-FSK (Audio FSK) tones.

TNC\_MODE\_1200\_AFSK,

/// 9600 bps using true FSK tones.

TNC\_MODE\_9600\_FSK

};

void tncInit();

bool\_t tncIsFree();

void tncHighRate(bool\_t state);

void tncSetMode (TNC\_DATA\_MODE dataMode);

void tnc1200TimerTick();

void tnc9600TimerTick();

void tncTxByte (uint8\_t value);

void tncTxPacket(TNC\_DATA\_MODE dataMode);

/\*\*

\* @defgroup ADC Analog To Digital Converter

\*

\* Control and manage the on board PIC A/D converter.

\*

\* @{

\*/

/// Filtered voltages using a single pole, low pass filter.

uint16\_t adcMainBusVolt;

/// PIC ADC Channel number of the reference voltage.

#define ADC\_REF 0

/// PIC ADC Channel number of the main bus voltage.

#define ADC\_MAINBUS 1

/// Input diode drop in units of 0.01 volts.

#define MAIN\_BUS\_VOLT\_OFFSET 20

/\*\*

\* Intialize the ADC subsystem.

\*/

void adcInit()

{

// Setup the ADC.

setup\_adc\_ports(AN0\_TO\_AN1);

setup\_adc( ADC\_CLOCK\_DIV\_32 );

// Zero the ADC filters.

adcMainBusVolt = 0;

}

/\*\*

\* Filtered main bus voltage in 10mV resolution.

\*

\* @return voltage in 10mV steps

\*/

uint16\_t adcGetMainBusVolt()

{

uint32\_t volts;

volts = (uint32\_t) (adcMainBusVolt >> 3);

volts = (volts \* 330l) / 1023l;

return (uint16\_t) volts + MAIN\_BUS\_VOLT\_OFFSET;

}

/\*\*

\* Get the current ADC value for the main bus voltage.

\*

\* @return ADC value in the range 0 to 1023

\*/

uint16\_t adcRawBusVolt()

{

set\_adc\_channel(ADC\_MAINBUS);

delay\_us(50);

return read\_adc();

}

/\*\*

\* Get the current ADC value for the reference source voltage.

\*

\* @return ADC value in the range 0 to 1023

\*/

uint16\_t adcRawRefVolt()

{

set\_adc\_channel(ADC\_REF);

delay\_us(50);

return read\_adc();

}

/\*\*

\* Read and filter the ADC channels for bus voltages.

\*/

void adcUpdate(void)

{

// Filter the bus voltage using a single pole low pass filter.

set\_adc\_channel(ADC\_MAINBUS);

delay\_us(50);

adcMainBusVolt = read\_adc() + adcMainBusVolt - (adcMainBusVolt >> 3);

}

/\*\* @} \*/

/\*\*

\* @defgroup diag Diagnostics and Control

\*

\* Functions for diagnostics and control of the hardware and flight data recorder.

\*

\* @{

\*/

/// Number of bytes per line to display when reading flight data recorder.

#define DIAG\_BYTES\_PER\_LINE 32

/\*\*

\* Process the command to erase the data logger flash.

\*/

void diagEraseFlash()

{

// Confirm we want to erase the flash with the key sequence 'yes' .

fprintf (PC\_HOST, "Are you sure (yes)? ");

if (fgetc(PC\_HOST) != 'y')

return;

if (fgetc(PC\_HOST) != 'e')

return;

if (fgetc(PC\_HOST) != 's')

return;

if (fgetc(PC\_HOST) != 13)

return;

// User feedback and erase the part.

fprintf (PC\_HOST, "Erasing flash...");

flashErase();

fprintf (PC\_HOST, "done.\n\r");

}

/\*\*

\* Display the engineering mode menu.

\*/

void diagMenu()

{

// User interface.

fprintf (PC\_HOST, "Options: (e)rase Flash, (r)ead Flash\n\r");

fprintf (PC\_HOST, " Toggle (L)ED\n\r");

fprintf (PC\_HOST, " (P)TT - Push To Transmit\n\r");

fprintf (PC\_HOST, " (f)requencey down, (F)requency up - 1KHz step\n\r");

fprintf (PC\_HOST, " (c)hannel down, (C)hannel up - 25KHz step\n\r");

fprintf (PC\_HOST, " (a)mplitude down, (A)mplitude up - 0.5 dB steps\n\r");

fprintf (PC\_HOST, " e(x)it engineering mode\n\r");

}

/\*\*

\* Process the command to dump the contents of the data logger flash.

\*/

void diagReadFlash()

{

bool\_t dataFoundFlag, userStopFlag;

uint8\_t i, buffer[DIAG\_BYTES\_PER\_LINE];

uint32\_t address;

// Set the initial conditions to read the flash.

address = 0x0000;

userStopFlag = false;

do

{

// Read each block from the flash device.

flashReadBlock (address, buffer, DIAG\_BYTES\_PER\_LINE);

// This flag will get set if any data byte is not equal to 0xff (erase flash state)

dataFoundFlag = false;

// Display the address.

fprintf (PC\_HOST, "%08lx ", address);

// Display each byte in the line.

for (i = 0; i < DIAG\_BYTES\_PER\_LINE; ++i)

{

fprintf (PC\_HOST, "%02x", buffer[i]);

// Set this flag if the cell is not erased.

if (buffer[i] != 0xff)

dataFoundFlag = true;

// Any key will abort the transfer.

if (kbhit(PC\_HOST))

userStopFlag = true;

} // END for

// at the end of each line.

fprintf (PC\_HOST, "\n\r");

// Advance to the next block of memory.

address += DIAG\_BYTES\_PER\_LINE;

} while (dataFoundFlag && !userStopFlag);

// Feedback to let the user know why the transfer stopped.

if (userStopFlag)

fprintf (PC\_HOST, "User aborted download!\n\r");

}

void diag1PPS()

{

uint16\_t timeStamp, lastTimeStamp;

lastTimeStamp = 0x0000;

gpsPowerOn();

for (;;)

{

timeStamp = CCP\_2;

if (timeStamp != lastTimeStamp)

{

delay\_ms (10);

timeStamp = CCP\_2;

fprintf (PC\_HOST, "%lu %lu\n\r", timeStamp, (timeStamp - lastTimeStamp));

lastTimeStamp = timeStamp;

}

}

}

/\*\*

\* Process diagnostic commands through the debug RS-232 port.

\*/

void diagPort()

{

bool\_t diagDoneFlag, ledFlag, paFlag, showSettingsFlag;

uint8\_t command, amplitude;

uint32\_t freqHz;

// If the input is low, we aren't connected to the RS-232 device so continue to boot.

if (!input(PIN\_B6))

return;

fprintf (PC\_HOST, "Engineering Mode\n\r");

fprintf (PC\_HOST, "Application Built %s %s\n\r", \_\_DATE\_\_, \_\_TIME\_\_);

// Current state of the status LED.

ledFlag = false;

output\_bit (IO\_LED, ledFlag);

// This flag indicates we are ready to leave the diagnostics mode.

diagDoneFlag = false;

// Current state of the PA.

paFlag = false;

// Flag that indicate we should show the current carrier frequency.

showSettingsFlag = false;

// Set the initial carrier frequency and amplitude.

freqHz = 445950000;

amplitude = 0;

// Wait for the exit command.

while (!diagDoneFlag)

{

// Wait for the user command.

command = fgetc(PC\_HOST);

// Decode and process the key stroke.

switch (command)

{

case 'e':

diagEraseFlash();

logInit();

break;

case 'l':

case 'L':

ledFlag = (ledFlag ? false : true);

output\_bit (IO\_LED, ledFlag);

break;

case 'h':

case 'H':

case '?':

diagMenu();

break;

case 'r':

diagReadFlash();

break;

case 't':

tncHighRate (true);

fprintf (PC\_HOST, "Set high rate TNC.\n\r");

break;

case 'f':

freqHz -= 1000;

ddsSetFreq (freqHz);

// Display the new frequency.

showSettingsFlag = true;

break;

case 'F':

freqHz += 1000;

ddsSetFreq (freqHz);

// Display the new frequency.

showSettingsFlag = true;

break;

case 'c':

freqHz -= 25000;

ddsSetFreq (freqHz);

// Display the new frequency.

showSettingsFlag = true;

break;

case 'C':

freqHz += 25000;

ddsSetFreq (freqHz);

// Display the new frequency.

showSettingsFlag = true;

break;

case 'p':

case 'P':

ddsSetFreq (freqHz);

paFlag = (paFlag ? false : true);

output\_bit (IO\_PTT, paFlag);

output\_bit (IO\_OSK, paFlag);

if (paFlag)

{

ddsSetMode (DDS\_MODE\_AFSK);

ddsSetAmplitude (amplitude);

} else

ddsSetMode (DDS\_MODE\_POWERDOWN);

break;

case 'a':

if (amplitude != 200)

{

amplitude += 5;

ddsSetAmplitude (amplitude);

// Display the new amplitude.

showSettingsFlag = true;

}

break;

case 'A':

if (amplitude != 0)

{

amplitude -= 5;

ddsSetAmplitude (amplitude);

// Display the new amplitude.

showSettingsFlag = true;

}

break;

case 'g':

diag1PPS();

break;

case 'x':

diagDoneFlag = true;

break;

default:

fprintf (PC\_HOST, "Invalid command. (H)elp for menu.\n\r");

break;

} // END switch

// Display the results of any user requests or commands.

if (showSettingsFlag)

{

showSettingsFlag = false;

fprintf (PC\_HOST, "%03ld.%03ld MHz ", freqHz / 1000000, (freqHz / 1000) % 1000);

fprintf (PC\_HOST, "%d.%01ddBc\n\r", amplitude / 10, amplitude % 10);

} // END if

} // END while

// Let the user know we are done with this mode.

fprintf (PC\_HOST, "Exit diagnostic mode.\n\r");

return;

}

/\*\* @} \*/

/\*\*

\* @defgroup DDS AD9954 DDS (Direct Digital Synthesizer)

\*

\* Functions to control the Analog Devices AD9954 DDS.

\*

\* @{

\*/

/// AD9954 CFR1 - Control functions including RAM, profiles, OSK, sync, sweep, SPI, and power control settings.

#define DDS\_AD9954\_CFR1 0x00

/// AD9954 CFR2 - Control functions including sync, PLL multiplier, VCO range, and charge pump current.

#define DDS\_AD9954\_CFR2 0x01

/// AD9954 ASF - Auto ramp rate speed control and output scale factor (0x0000 to 0x3fff).

#define DDS\_AD9954\_ASF 0x02

/// AD9954 ARR - Amplitude ramp rate for OSK function.

#define DDS\_AD9954\_ARR 0x03

/// AD9954 FTW0 - Frequency tuning word 0.

#define DDS\_AD9954\_FTW0 0x04

/// AD9954 FTW1 - Frequency tuning word 1

#define DDS\_AD9954\_FTW1 0x06

/// AD9954 NLSCW - Negative Linear Sweep Control Word used for spectral shaping in FSK mode

#define DDS\_AD9954\_NLSCW 0x07

/// AD9954 PLSCW - Positive Linear Sweep Control Word used for spectral shaping in FSK mode

#define DDS\_AD9954\_PLSCW 0x08

/// AD9954 RSCW0 - RAM Segment Control Word 0

#define DDS\_AD9954\_RWCW0 0x07

/// AD9954 RSCW0 - RAM Segment Control Word 1

#define DDS\_AD9954\_RWCW1 0x08

/// AD9954 RAM segment

#define DDS\_RAM 0x0b

/// Current operational mode.

DDS\_MODE ddsMode;

/// Number of digits in DDS frequency to FTW conversion.

#define DDS\_FREQ\_TO\_FTW\_DIGITS 9

/// Array of multiplication factors used to convert frequency to the FTW.

const uint32\_t DDS\_MULT[DDS\_FREQ\_TO\_FTW\_DIGITS] = { 11, 7, 7, 3, 4, 8, 4, 9, 1 };

/// Array of divisors used to convert frequency to the FTW.

const uint32\_t DDS\_DIVISOR[DDS\_FREQ\_TO\_FTW\_DIGITS - 1] = { 10, 100, 1000, 10000, 100000, 1000000, 10000000, 100000000 };

/// Lookup table to convert dB amplitude scale in 0.5 steps to a linear DDS scale factor.

const uint16\_t DDS\_AMP\_TO\_SCALE[] =

{

16383, 15467, 14601, 13785, 13013, 12286, 11598, 10949, 10337, 9759, 9213, 8697,

8211, 7752, 7318, 6909, 6522, 6157, 5813, 5488, 5181, 4891, 4617, 4359, 4115, 3885, 3668, 3463,

3269, 3086, 2913, 2750, 2597, 2451, 2314, 2185, 2062, 1947, 1838, 1735, 1638

};

/// Frequency Word List - 4.0KHz FM frequency deviation at 81.15MHz (445.950MHz)

const uint32\_t freqTable[256] =

{

955418300, 955419456, 955420611, 955421765, 955422916, 955424065, 955425210, 955426351,

955427488, 955428618, 955429743, 955430861, 955431971, 955433073, 955434166, 955435249,

955436322, 955437385, 955438435, 955439474, 955440500, 955441513, 955442511, 955443495,

955444464, 955445417, 955446354, 955447274, 955448176, 955449061, 955449926, 955450773,

955451601, 955452408, 955453194, 955453960, 955454704, 955455426, 955456126, 955456803,

955457457, 955458088, 955458694, 955459276, 955459833, 955460366, 955460873, 955461354,

955461809, 955462238, 955462641, 955463017, 955463366, 955463688, 955463983, 955464250,

955464489, 955464701, 955464884, 955465040, 955465167, 955465266, 955465337, 955465380,

955465394, 955465380, 955465337, 955465266, 955465167, 955465040, 955464884, 955464701,

955464489, 955464250, 955463983, 955463688, 955463366, 955463017, 955462641, 955462238,

955461809, 955461354, 955460873, 955460366, 955459833, 955459276, 955458694, 955458088,

955457457, 955456803, 955456126, 955455426, 955454704, 955453960, 955453194, 955452408,

955451601, 955450773, 955449926, 955449061, 955448176, 955447274, 955446354, 955445417,

955444464, 955443495, 955442511, 955441513, 955440500, 955439474, 955438435, 955437385,

955436322, 955435249, 955434166, 955433073, 955431971, 955430861, 955429743, 955428618,

955427488, 955426351, 955425210, 955424065, 955422916, 955421765, 955420611, 955419456,

955418300, 955417144, 955415989, 955414836, 955413684, 955412535, 955411390, 955410249,

955409113, 955407982, 955406857, 955405740, 955404629, 955403528, 955402435, 955401351,

955400278, 955399216, 955398165, 955397126, 955396100, 955395088, 955394089, 955393105,

955392136, 955391183, 955390246, 955389326, 955388424, 955387540, 955386674, 955385827,

955385000, 955384192, 955383406, 955382640, 955381896, 955381174, 955380474, 955379797,

955379143, 955378513, 955377906, 955377324, 955376767, 955376235, 955375728, 955375246,

955374791, 955374362, 955373959, 955373583, 955373234, 955372912, 955372618, 955372350,

955372111, 955371900, 955371716, 955371560, 955371433, 955371334, 955371263, 955371220,

955371206, 955371220, 955371263, 955371334, 955371433, 955371560, 955371716, 955371900,

955372111, 955372350, 955372618, 955372912, 955373234, 955373583, 955373959, 955374362,

955374791, 955375246, 955375728, 955376235, 955376767, 955377324, 955377906, 955378513,

955379143, 955379797, 955380474, 955381174, 955381896, 955382640, 955383406, 955384192,

955385000, 955385827, 955386674, 955387540, 955388424, 955389326, 955390246, 955391183,

955392136, 955393105, 955394089, 955395088, 955396100, 955397126, 955398165, 955399216,

955400278, 955401351, 955402435, 955403528, 955404629, 955405740, 955406857, 955407982,

955409113, 955410249, 955411390, 955412535, 955413684, 955414836, 955415989, 955417144

};

/\*\*

\* Initialize the DDS regsiters and RAM.

\*/

void ddsInit()

{

// Setup the SPI port for the DDS interface.

setup\_spi( SPI\_MASTER | SPI\_L\_TO\_H | SPI\_CLK\_DIV\_4 | SPI\_XMIT\_L\_TO\_H );

// Set the initial DDS mode. The ddsSetMode function uses this value to make the desired DDS selections.

ddsMode = DDS\_MODE\_NOT\_INITIALIZED;

// Set the DDS operational mode.

ddsSetMode (DDS\_MODE\_POWERDOWN);

// Set the output to full scale.

ddsSetOutputScale (0x3fff);

// CFR2 (Control Function Register No. 2)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_CFR2);

spi\_write (0x00); // Unused register bits

spi\_write (0x00);

spi\_write (0x9c); // 19x reference clock multipler, high VCO range, nominal charge pump current

output\_high (IO\_CS);

// ARR (Amplitude Ramp Rate) to 15mS for OSK

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_ARR);

spi\_write (83);

output\_high (IO\_CS);

// Strobe the part so we apply the updates.

output\_high (IO\_UPDATE);

output\_low (IO\_UPDATE);

}

/\*\*

\* Set DDS amplitude value in the range 0 to 16383 where 16383 is full scale. This value is a

\* linear multiplier and needs to be scale for RF output power in log scale.

\*

\* @param scale in the range 0 to 16383

\*/

void ddsSetOutputScale (uint16\_t scale)

{

// Set ASF (Amplitude Scale Factor)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_ASF);

spi\_write ((scale >> 8) & 0xff);

spi\_write (scale & 0xff);

output\_high (IO\_CS);

// Strobe the DDS to set the amplitude.

output\_high (IO\_UPDATE);

output\_low (IO\_UPDATE);

}

/\*\*

\* Set the DDS amplitude in units of dBc of full scale where 1 is 0.1 dB. For example, a value of 30 is 3dBc

\* or a value of 85 is 8.5dBc.

\*

\* @param amplitude in 0.1 dBc of full scale

\*/

void ddsSetAmplitude (uint8\_t amplitude)

{

// Range limit based on the lookup table size.

if (amplitude > 200)

return;

// Set the linear DDS ASF (Amplitude Scale Factor) based on the dB lookup table.

ddsSetOutputScale (DDS\_AMP\_TO\_SCALE[amplitude / 5]);

// Toggle the DDS output low and then high to force it to ramp to the new output level setting.

output\_low (IO\_OSK);

delay\_ms(25);

output\_high (IO\_OSK);

delay\_ms(25);

}

/\*\*

\* Set DDS frequency tuning word. The output frequency is equal to RefClock \* (ftw / 2 ^ 32).

\*

\* @param ftw Frequency Tuning Word

\*/

void ddsSetFTW (uint32\_t ftw)

{

// Set FTW0 (Frequency Tuning Word 0)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_FTW0);

spi\_write ((ftw >> 24) & 0xff);

spi\_write ((ftw >> 16) & 0xff);

spi\_write ((ftw >> 8) & 0xff);

spi\_write (ftw & 0xff);

output\_high (IO\_CS);

// Strobe the DDS to set the frequency.

output\_high (IO\_UPDATE);

output\_low (IO\_UPDATE);

}

/\*\*

\* Convert frequency in hertz to 32-bit DDS FTW (Frequency Tune Word).

\*

\* @param freq frequency in Hertz

\*

\*/

void ddsSetFreq(uint32\_t freq)

{

uint8\_t i;

uint32\_t ftw;

// To avoid rounding errors with floating point math, we do a long multiply on the data.

ftw = freq \* DDS\_MULT[0];

for (i = 0; i < DDS\_FREQ\_TO\_FTW\_DIGITS - 1; ++i)

ftw += (freq \* DDS\_MULT[i+1]) / DDS\_DIVISOR[i];

ddsSetFTW (ftw);

}

/\*\*

\* Set DDS frequency tuning word for the FSK 0 and 1 values. The output frequency is equal

\* to RefClock \* (ftw / 2 ^ 32).

\*

\* @param ftw0 frequency tuning word for the FSK 0 value

\* @param ftw1 frequency tuning word for the FSK 1 value

\*/

void ddsSetFSKFreq (uint32\_t ftw0, uint32\_t ftw1)

{

// Set FTW0 (Frequency Tuning Word 0)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_FTW0);

spi\_write ((ftw0 >> 24) & 0xff);

spi\_write ((ftw0 >> 16) & 0xff);

spi\_write ((ftw0 >> 8) & 0xff);

spi\_write (ftw0 & 0xff);

output\_high (IO\_CS);

// Set FTW0 (Frequency Tuning Word 1)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_FTW1);

spi\_write ((ftw1 >> 24) & 0xff);

spi\_write ((ftw1 >> 16) & 0xff);

spi\_write ((ftw1 >> 8) & 0xff);

spi\_write (ftw1 & 0xff);

output\_high (IO\_CS);

// Strobe the DDS to set the frequency.

output\_high (IO\_UPDATE);

output\_low (IO\_UPDATE);

}

/\*\*

\* Set the DDS to run in A-FSK, FSK, or PSK31 mode

\*

\* @param mode *DDS\_MODE\_APRS*, *DDS\_MODE\_PSK31*, or *DDS\_MODE\_HF\_APRS* constant

\*/

void ddsSetMode (DDS\_MODE mode)

{

// Save the current mode.

ddsMode = mode;

switch (mode)

{

case DDS\_MODE\_POWERDOWN:

// CFR1 (Control Function Register No. 1)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_CFR1);

spi\_write (0x00);

spi\_write (0x00);

spi\_write (0x00);

spi\_write (0xf0); // Power down all subsystems.

output\_high (IO\_CS);

break;

case DDS\_MODE\_AFSK:

// CFR1 (Control Function Register No. 1)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_CFR1);

spi\_write (0x03); // OSK Enable and Auto OSK keying

spi\_write (0x00);

spi\_write (0x00);

spi\_write (0x40); // Power down comparator circuit

output\_high (IO\_CS);

break;

case DDS\_MODE\_FSK:

// CFR1 (Control Function Register No. 1)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_CFR1);

spi\_write (0x03); // Clear RAM Enable, OSK Enable, Auto OSK keying

spi\_write (0x00);

spi\_write (0x00);

spi\_write (0x40); // Power down comparator circuit

output\_high (IO\_CS);

// NOTE: The sweep rate requires 1/4 of a bit time (26uS) to transition.

// 6KHz delta = 70641 counts = (6KHz / 364.8MHz) \* 2 ^ 32

// SYNC\_CLK = 91.2MHz 1/91.2MHz \* 70641 \* 1/29 = 26.7uS

// NLSCW (Negative Linear Sweep Control Word)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_NLSCW);

spi\_write (1); // Falling sweep ramp rate word

spi\_write (0x00); // Delta frequency tuning word

spi\_write (0x00);

spi\_write (0x00);

spi\_write (250);

output\_high (IO\_CS);

// PLSCW (Positive Linear Sweep Control Word)

output\_low (IO\_CS);

spi\_write (DDS\_AD9954\_PLSCW);

spi\_write (1); // Rising sweep ramp rate word

spi\_write (0x00); // Delta frequency tuning word

spi\_write (0x00);

spi\_write (0x00);

spi\_write (250);

output\_high (IO\_CS);

break;

} // END switch

// Strobe the DDS to change the mode.

output\_high (IO\_UPDATE);

output\_low (IO\_UPDATE);

}

/\*\* @} \*/

/\*\*

\* @defgroup flash Flash Manager

\*

\* Functions to control the ST MP25P80 serial flash device.

\*

\* @{

\*/

/// Flash Chip Select - Port B3

#define FLASH\_CS PIN\_B3

/// Flash Clock - Port B5

#define FLASH\_CLK PIN\_B5

/// Flash Data Input - Port B4

#define FLASH\_D PIN\_B4

/// Flash Data Output - Port B2

#define FLASH\_Q PIN\_B2

/\*\*

\* Determine if a flash write or erase operation is currently in progress.

\*

\* @return true if write/erase in progress

\*/

bool\_t flashIsWriteInProgress()

{

uint8\_t status;

output\_low (FLASH\_CS);

// Read Status Register (RDSR) flash command.

flashSendByte (0x05);

status = flashGetByte();

output\_high (FLASH\_CS);

return (((status & 0x01) == 0x01) ? true : false);

}

/\*\*

\* Read a block of memory from the flash device.

\*

\* @param address of desired location in the range 0x00000 to 0xFFFFF (1MB)

\* @param block pointer to locate of data block

\* @param length number of bytes to read

\*/

void flashReadBlock(uint32\_t address, uint8\_t \*block, uint16\_t length)

{

uint16\_t i;

output\_low (FLASH\_CS);

// Read Data Byte(s) (READ) flash command.

flashSendByte (0x03);

flashSendAddress (address);

for (i = 0; i < length; ++i)

\*block++ = flashGetByte();

output\_high (FLASH\_CS);

}

/\*\*

\* Write a block of memory to the flash device.

\*

\* @param address of desired location in the range 0x00000 to 0xFFFFF (1MB)

\* @param block pointer data block to write

\* @param length number of bytes to write

\*/

void flashWriteBlock(uint32\_t address, uint8\_t \*block, uint8\_t length)

{

uint8\_t i;

output\_low (FLASH\_CS);

// Write Enable (WREN) flash command.

flashSendByte (0x06);

output\_high (FLASH\_CS);

output\_low (FLASH\_CS);

// Page Program (PP) flash command.

flashSendByte (0x02);

flashSendAddress (address);

for (i = 0; i < length; ++i)

{

// Send each byte in the data block.

flashSendByte (\*block++);

// Track the address in the flash device.

++address;

// If we cross a page boundary (a page is 256 bytes) we need to stop and send the address again.

if ((address & 0xff) == 0x00)

{

output\_high (FLASH\_CS);

// Write this block of data.

while (flashIsWriteInProgress());

output\_low (FLASH\_CS);

// Write Enable (WREN) flash command.

flashSendByte (0x06);

output\_high (FLASH\_CS);

output\_low (FLASH\_CS);

// Page Program (PP) flash command.

flashSendByte (0x02);

flashSendAddress (address);

} // END if

} // END for

output\_high (FLASH\_CS);

// Wait for the final write operation to complete.

while (flashIsWriteInProgress());

}

/\*\*

\* Erase the entire flash device (all locations set to 0xff).

\*/

void flashErase()

{

output\_low (FLASH\_CS);

// Write Enable (WREN) flash command.

flashSendByte (0x06);

output\_high (FLASH\_CS);

output\_low (FLASH\_CS);

// Bulk Erase (BE) flash command.

flashSendByte (0xc7);

output\_high (FLASH\_CS);

while (flashIsWriteInProgress());

}

/\*\*

\* Read a single byte from the flash device through the serial interface. This function

\* only controls the clock line. The chip select must be configured before calling

\* this function.

\*

\* @return byte read from device

\*/

uint8\_t flashGetByte()

{

uint8\_t i, value;

value = 0;

// Bit bang the 8-bits.

for (i = 0; i < 8; ++i)

{

// Data is ready on the rising edge of the clock.

output\_high (FLASH\_CLK);

// MSB is first, so shift left.

value = value << 1;

if (input (FLASH\_Q))

value = value | 0x01;

output\_low (FLASH\_CLK);

} // END for

return value;

}

/\*\*

\* Initialize the flash memory subsystem.

\*/

void flashInit()

{

// I/O lines to control flash.

output\_high (FLASH\_CS);

output\_low (FLASH\_CLK);

output\_low (FLASH\_D);

}

/\*\*

\* Write a single byte to the flash device through the serial interface. This function

\* only controls the clock line. The chip select must be configured before calling

\* this function.

\*

\* @param value byte to write to device

\*/

void flashSendByte(uint8\_t value)

{

uint8\_t i;

// Bit bang the 8-bits.

for (i = 0; i < 8; ++i)

{

// Drive the data input pin.

if ((value & 0x80) == 0x80)

output\_high (FLASH\_D);

else

output\_low (FLASH\_D);

// MSB is first, so shift leeft.

value = value << 1;

// Data is accepted on the rising edge of the clock.

output\_high (FLASH\_CLK);

output\_low (FLASH\_CLK);

} // END for

}

/\*\*

\* Write the 24-bit address to the flash device through the serial interface. This function

\* only controls the clock line. The chip select must be configured before calling

\* this function.

\*

\* @param address 24-bit flash device address

\*/

void flashSendAddress(uint32\_t address)

{

uint8\_t i;

// Bit bang the 24-bits.

for (i = 0; i < 24; ++i)

{

// Drive the data input pin.

if ((address & 0x800000) == 0x800000)

output\_high (FLASH\_D);

else

output\_low (FLASH\_D);

// MSB is first, so shift left.

address = address << 1;

// Data is accepted on the rising edge of the clock.

output\_high (FLASH\_CLK);

output\_low (FLASH\_CLK);

} // END for

}

/\*\* @} \*/

/\*\*

\* @defgroup GPS Motorola M12+ GPS Engine

\*

\* Functions to control the Motorola M12+ GPS engine in native binary protocol mode.

\*

\* @{

\*/

/// The maximum length of a binary GPS engine message.

#define GPS\_BUFFER\_SIZE 50

/// GPS parse engine state machine values.

enum GPS\_PARSE\_STATE\_MACHINE

{

/// 1st start character '@'

GPS\_START1,

/// 2nd start character '@'

GPS\_START2,

/// Upper case 'A' - 'Z' message type

GPS\_COMMAND1,

/// Lower case 'a' - 'z' message type

GPS\_COMMAND2,

/// 0 - xx bytes based on message type 'Aa'

GPS\_READMESSAGE,

/// 8-bit checksum

GPS\_CHECKSUMMESSAGE,

/// End of message - Carriage Return

GPS\_EOMCR,

/// End of message - Line Feed

GPS\_EOMLF

};

/// Index into gpsBuffer used to store message data.

uint8\_t gpsIndex;

/// State machine used to parse the GPS message stream.

GPS\_PARSE\_STATE\_MACHINE gpsParseState;

/// Buffer to store data as it is read from the GPS engine.

uint8\_t gpsBuffer[GPS\_BUFFER\_SIZE];

/// Peak altitude detected while GPS is in 3D fix mode.

int32\_t gpsPeakAltitude;

/// Checksum used to verify binary message from GPS engine.

uint8\_t gpsChecksum;

/// Last verified GPS message received.

GPSPOSITION\_STRUCT gpsPosition;

/\*\*

\* Get the type of fix.

\*

\* @return gps fix type enumeration

\*/

GPS\_FIX\_TYPE gpsGetFixType()

{

// The upper 3-bits determine the fix type.

switch (gpsPosition.status & 0xe000)

{

case 0xe000:

return GPS\_3D\_FIX;

case 0xc000:

return GPS\_2D\_FIX;

default:

return GPS\_NO\_FIX;

} // END switch

}

/\*\*

\* Peak altitude detected while GPS is in 3D fix mode since the system was booted.

\*

\* @return altitude in feet

\*/

int32\_t gpsGetPeakAltitude()

{

return gpsPeakAltitude;

}

/\*\*

\* Initialize the GPS subsystem.

\*/

void gpsInit()

{

// Initial parse state.

gpsParseState = GPS\_START1;

// Assume we start at sea level.

gpsPeakAltitude = 0;

// Clear the structure that stores the position message.

memset (&gpsPosition, 0, sizeof(GPSPOSITION\_STRUCT));

// Setup the timers used to measure the 1-PPS time period.

setup\_timer\_3(T3\_INTERNAL | T3\_DIV\_BY\_1);

setup\_ccp2 (CCP\_CAPTURE\_RE | CCP\_USE\_TIMER3);

}

/\*\*

\* Determine if new GPS message is ready to process. This function is a one shot and

\* typically returns true once a second for each GPS position fix.

\*

\* @return true if new message available; otherwise false

\*/

bool\_t gpsIsReady()

{

if (gpsPosition.updateFlag)

{

gpsPosition.updateFlag = false;

return true;

} // END if

return false;

}

/\*\*

\* Calculate NMEA-0183 message checksum of **buffer** that is **length** bytes long.

\*

\* @param buffer pointer to data buffer.

\* @param length number of bytes in buffer.

\*

\* @return checksum of buffer

\*/

uint8\_t gpsNMEAChecksum (uint8\_t \*buffer, uint8\_t length)

{

uint8\_t i, checksum;

checksum = 0;

for (i = 0; i < length; ++i)

checksum ^= buffer[i];

return checksum;

}

/\*\*

\* Verify the GPS engine is sending the @@Hb position report message. If not,

\* configure the GPS engine to send the desired report.

\*

\* @return true if GPS engine operation; otherwise false

\*/

bool\_t gpsSetup()

{

uint8\_t startTime, retryCount;

// We wait 10 seconds for the GPS engine to respond to our message request.

startTime = timeGetTicks();

retryCount = 0;

while (++retryCount < 10)

{

// Read the serial FIFO and process the GPS messages.

gpsUpdate();

// If a GPS data set is available, then GPS is operational.

if (gpsIsReady())

{

timeSetDutyCycle (TIME\_DUTYCYCLE\_10);

return true;

}

if (timeGetTicks() > startTime)

{

puts ("@@Hb\001\053\015\012");

startTime += 10;

} // END if

} // END while

return false;

}

/\*\*

\* Parse the Motorola @@Hb (Short position/message) report.

\*/

void gpsParsePositionMessage()

{

// Convert the binary stream into data elements. We will scale to the desired units

// as the values are used.

gpsPosition.updateFlag = true;

gpsPosition.month = gpsBuffer[0];

gpsPosition.day = gpsBuffer[1];

gpsPosition.year = ((uint16\_t) gpsBuffer[2] << 8) | gpsBuffer[3];

gpsPosition.hours = gpsBuffer[4];

gpsPosition.minutes = gpsBuffer[5];

gpsPosition.seconds = gpsBuffer[6];

gpsPosition.latitude = ((int32) gpsBuffer[11] << 24) | ((int32) gpsBuffer[12] << 16) | ((int32) gpsBuffer[13] << 8) | (int32) gpsBuffer[14];

gpsPosition.longitude = ((int32) gpsBuffer[15] << 24) | ((int32) gpsBuffer[16] << 16) | ((int32) gpsBuffer[17] << 8) | gpsBuffer[18];

gpsPosition.altitudeCM = ((int32) gpsBuffer[19] << 24) | ((int32) gpsBuffer[20] << 16) | ((int32) gpsBuffer[21] << 8) | gpsBuffer[22];

gpsPosition.altitudeFeet = gpsPosition.altitudeCM \* 100l / 3048l;

gpsPosition.vSpeed = ((uint16\_t) gpsBuffer[27] << 8) | gpsBuffer[28];

gpsPosition.hSpeed = ((uint16\_t) gpsBuffer[29] << 8) | gpsBuffer[30];

gpsPosition.heading = ((uint16\_t) gpsBuffer[31] << 8) | gpsBuffer[32];

gpsPosition.dop = ((uint16\_t) gpsBuffer[33] << 8) | gpsBuffer[34];

gpsPosition.visibleSats = gpsBuffer[35];

gpsPosition.trackedSats = gpsBuffer[36];

gpsPosition.status = ((uint16\_t) gpsBuffer[37] << 8) | gpsBuffer[38];

// Update the peak altitude if we have a valid 3D fix.

if (gpsGetFixType() == GPS\_3D\_FIX)

if (gpsPosition.altitudeFeet > gpsPeakAltitude)

gpsPeakAltitude = gpsPosition.altitudeFeet;

}

/\*\*

\* Turn on the GPS engine power and serial interface.

\*/

void gpsPowerOn()

{

// 3.0 VDC LDO control line.

output\_high (IO\_GPS\_PWR);

// Enable the UART and the transmit line.

#asm

bsf 0xFAB.7

#endasm

}

/\*\*

\* Turn off the GPS engine power and serial interface.

\*/

void gpsPowerOff()

{

// Disable the UART and the transmit line.

#asm

bcf 0xFAB.7

#endasm

// 3.0 VDC LDO control line.

output\_low (IO\_GPS\_PWR);

}

/\*\*

\* Read the serial FIFO and process complete GPS messages.

\*/

void gpsUpdate()

{

uint8\_t value;

// This state machine handles each characters as it is read from the GPS serial port.

// We are looking for the GPS mesage @@Hb ... C

while (serialHasData())

{

// Get the character value.

value = serialRead();

// Process based on the state machine.

switch (gpsParseState)

{

case GPS\_START1:

if (value == '@')

gpsParseState = GPS\_START2;

break;

case GPS\_START2:

if (value == '@')

gpsParseState = GPS\_COMMAND1;

else

gpsParseState = GPS\_START1;

break;

case GPS\_COMMAND1:

if (value == 'H')

gpsParseState = GPS\_COMMAND2;

else

gpsParseState = GPS\_START1;

break;

case GPS\_COMMAND2:

if (value == 'b')

{

gpsParseState = GPS\_READMESSAGE;

gpsIndex = 0;

gpsChecksum = 0;

gpsChecksum ^= 'H';

gpsChecksum ^= 'b';

} else

gpsParseState = GPS\_START1;

break;

case GPS\_READMESSAGE:

gpsChecksum ^= value;

gpsBuffer[gpsIndex++] = value;

if (gpsIndex == 47)

gpsParseState = GPS\_CHECKSUMMESSAGE;

break;

case GPS\_CHECKSUMMESSAGE:

if (gpsChecksum == value)

gpsParseState = GPS\_EOMCR;

else

gpsParseState = GPS\_START1;

break;

case GPS\_EOMCR:

if (value == 13)

gpsParseState = GPS\_EOMLF;

else

gpsParseState = GPS\_START1;

break;

case GPS\_EOMLF:

// Once we have the last character, convert the binary message to something usable.

if (value == 10)

gpsParsePositionMessage();

gpsParseState = GPS\_START1;

break;

} // END switch

} // END while

}

/\*\* @} \*/

/\*\*

\* @defgroup log Flight Data Recorder

\*

\* Functions to manage and control the flight data recorder

\*

\* @{

\*/

/// Number of bytes to buffer before writing to flash memory.

#define LOG\_WRITE\_BUFFER\_SIZE 40

/// Last used address in flash memory.

uint32\_t logAddress;

/// Temporary buffer that holds data before it is written to flash device.

uint8\_t logBuffer[LOG\_WRITE\_BUFFER\_SIZE];

/// Current index into log buffer.

uint8\_t logIndex;

/\*\*

\* Last used address in flash memory. This location is where the next log data will

\* be written.

\*

\* @return 24-bit flash memory address

\*/

uint32\_t logGetAddress()

{

return logAddress;

}

/\*\*

\* Write the contents of the temporary log buffer to the flash device. If the buffer

\* is empty, nothing is done.

\*/

void logFlush()

{

// We only need to write if there is data.

if (logIndex != 0)

{

flashWriteBlock (logAddress, logBuffer, logIndex);

logAddress += logIndex;

logIndex = 0;

} // END if

}

/\*\*

\* Prepare the flight data recorder for logging.

\*/

void logInit()

{

uint8\_t buffer[8];

bool\_t endFound;

fprintf (PC\_HOST, "Searching for end of flash log...");

logAddress = 0x0000;

endFound = false;

// Read each logged data block from flash to determine how long it is.

do

{

// Read the data log entry type.

flashReadBlock (logAddress, buffer, 1);

// Based on the log entry type, we'll skip over the data contained in the entry.

switch (buffer[0])

{

case LOG\_BOOTED:

logAddress += 7;

break;

case LOG\_COORD:

logAddress += 26;

break;

case LOG\_TEMPERATURE:

logAddress += 3;

break;

case LOG\_VOLTAGE:

logAddress += 5;

break;

case 0xff:

endFound = true;

break;

default:

++logAddress;

} // END switch

} while (logAddress < 0x100000 && !endFound);

fprintf (PC\_HOST, "done. Log contains %ld bytes.\n\r", logAddress);

logIndex = 0;

}

/\*\*

\* Start a entry in the data log.

\*

\* @param type of log entry, i.e. LOG\_BOOTED, LOG\_COORD, etc.

\*/

void logType (LOG\_TYPE type)

{

// Only add the new entry if there is space.

if (logAddress >= 0x100000)

return;

// Write the old entry first.

logFlush();

// Save the type and set the log buffer pointer.

logBuffer[0] = type;

logIndex = 1;

}

/\*\*

\* Save an unsigned, 8-bit value in the log.

\*

\* @param value unsigned, 8-bit value

\*/

void logUint8 (uint8\_t value)

{

logBuffer[logIndex++] = value;

}

/\*\*

\* Save a signed, 16-bit value in the log.

\*

\* @param value signed, 16-bit value

\*/

void logInt16 (int16\_t value)

{

logBuffer[logIndex++] = (value >> 8) & 0xff;

logBuffer[logIndex++] = value & 0xff;

}

/\*\*

\* Save an unsigned, 16-bit value in the log.

\*

\* @param value unsigned, 16-bit value

\*/

void logUint16 (uint16\_t value)

{

logBuffer[logIndex++] = (value >> 8) & 0xff;

logBuffer[logIndex++] = value & 0xff;

}

/\*\*

\* Save a signed, 32-bit value in the log.

\*

\* @param value signed, 32-bit value

\*/

void logInt32 (int32\_t value)

{

logBuffer[logIndex++] = (value >> 24) & 0xff;

logBuffer[logIndex++] = (value >> 16) & 0xff;

logBuffer[logIndex++] = (value >> 8) & 0xff;

logBuffer[logIndex++] = value & 0xff;

}

/\*\* @} \*/

/\*\*

\* @defgroup LM92 LM92 temperature sensor

\*

\* Read and control the National Semiconductor LM92 I2C temperature sensor

\*

\* @{

\*/

/\*\*

\* Read the LM92 temperature value in 0.1 degrees F.

\*

\* @return 0.1 degrees F

\*/

int16\_t lm92GetTemp()

{

int16\_t value;

int32\_t temp;

// Set the SDA and SCL to input pins to control the LM92.

set\_tris\_c (0x9a);

// Read the temperature register value.

i2c\_start();

i2c\_write(0x97);

value = ((int16\_t) i2c\_read() << 8);

value = value | ((int16\_t) i2c\_read() & 0x00f8);

i2c\_stop();

// Set the SDA and SCL back to outputs for use with the AD9954 because we share common clock pins.

set\_tris\_c (0x82);

// LM92 register 0.0625degC/bit 9 10 9

// ------------- \* -------------- \* - \* -- = -- + 320

// 8 5 64

// Convert to degrees F.

temp = (int32\_t) value;

temp = ((temp \* 9l) / 64l) + 320;

return (int16\_t) temp;

}

/\*\* @} \*/

/\*\*

\* @defgroup serial Serial Port FIFO

\*

\* FIFO for the built-in serial port.

\*

\* @{

\*/

/// Size of serial port FIFO in bytes. It must be a power of 2, i.e. 2, 4, 8, 16, etc.

#define SERIAL\_BUFFER\_SIZE 64

/// Mask to wrap around at end of circular buffer. (SERIAL\_BUFFER\_SIZE - 1)

#define SERIAL\_BUFFER\_MASK 0x3f

/// Index to the next free location in the buffer.

uint8\_t serialHead;

/// Index to the next oldest data in the buffer.

uint8\_t serialTail;

/// Circular buffer (FIFO) to hold serial data.

uint8\_t serialBuffer[SERIAL\_BUFFER\_SIZE];

/\*\*

\* Determine if the FIFO contains data.

\*

\* @return true if data present; otherwise false

\*/

bool\_t serialHasData()

{

if (serialHead == serialTail)

return false;

return true;

}

/\*\*

\* Initialize the serial processor.

\*/

void serialInit()

{

serialHead = 0;

serialTail = 0;

}

/\*\*

\* Get the oldest character from the FIFO.

\*

\* @return oldest character; 0 if FIFO is empty

\*/

uint8\_t serialRead()

{

uint8\_t value;

// Make sure we have something to return.

if (serialHead == serialTail)

return 0;

// Save the value.

value = serialBuffer[serialTail];

// Update the pointer.

serialTail = (serialTail + 1) & SERIAL\_BUFFER\_MASK;

return value;

}

/\*\*

\* Read and store any characters in the PIC serial port in a FIFO.

\*/

void serialUpdate()

{

// If there isn't a character in the PIC buffer, just leave.

while (kbhit())

{

// Save the value in the FIFO.

serialBuffer[serialHead] = getc();

// Move the pointer to the next open space.

serialHead = (serialHead + 1) & SERIAL\_BUFFER\_MASK;

}

}

/\*\* @} \*/

/\*\*

\* @defgroup sys System Library Functions

\*

\* Generic system functions similiar to the run-time C library.

\*

\* @{

\*/

/\*\*

\* Calculate the CRC-16 CCITT of **buffer** that is **length** bytes long.

\* The **crc** parameter allow the calculation on the CRC on multiple buffers.

\*

\* @param buffer Pointer to data buffer.

\* @param length number of bytes in data buffer

\* @param crc starting value

\*

\* @return CRC-16 of buffer[0 .. length]

\*/

uint16\_t sysCRC16(uint8\_t \*buffer, uint8\_t length, uint16\_t crc)

{

uint8\_t i, bit, value;

for (i = 0; i < length; ++i)

{

value = buffer[i];

for (bit = 0; bit < 8; ++bit)

{

crc ^= (value & 0x01);

crc = ( crc & 0x01 ) ? ( crc >> 1 ) ^ 0x8408 : ( crc >> 1 );

value = value >> 1;

} // END for

} // END for

return crc ^ 0xffff;

}

/\*\*

\* Initialize the system library and global resources.

\*/

void sysInit()

{

gpsPowerOff ();

output\_high (IO\_LED);

output\_high (IO\_CS);

output\_low (IO\_PS1);

output\_low (IO\_PS0);

output\_low (IO\_OSK);

output\_low (IO\_UPDATE);

output\_low (IO\_PTT);

output\_low (IO\_GPS\_TXD);

// Configure the port direction (input/output).

set\_tris\_a (0xc3);

set\_tris\_b (0x44);

set\_tris\_c (0x82);

// Display a startup message during boot.

fprintf (PC\_HOST, "System booted.\n\r");

}

/\*\*

\* Log the current GPS position.

\*/

void sysLogGPSData()

{

// Log the data.

logType (LOG\_COORD);

logUint8 (gpsPosition.hours);

logUint8 (gpsPosition.minutes);

logUint8 (gpsPosition.seconds);

logInt32 (gpsPosition.latitude);

logInt32 (gpsPosition.longitude);

logInt32 (gpsPosition.altitudeCM);

logUint16 (gpsPosition.vSpeed);

logUint16 (gpsPosition.hSpeed);

logUint16 (gpsPosition.heading);

logUint16 (gpsPosition.status);

logUint8 ((uint8\_t) (gpsPosition.dop & 0xff));

logUint8 ((uint8\_t) ((gpsPosition.visibleSats << 4) | gpsPosition.trackedSats));

}

/\*\*

\* Log the ADC values of the bus and reference voltage values.

\*/

void sysLogVoltage()

{

logType (LOG\_VOLTAGE);

logUint16 (adcRawBusVolt());

logUint16 (adcRawRefVolt());

}

/\*\* @} \*/

/\*\*

\* @defgroup rtc Real Time Interrupt tick

\*

\* Manage the built-in real time interrupt. The interrupt clock PRI is 104uS (9600 bps).

\*

\* @{

\*/

/// A counter that ticks every 100mS.

uint8\_t timeTicks;

/// Counts the number of 104uS interrupts for a 100mS time period.

uint16\_t timeInterruptCount;

/// Counts the number of 100mS time periods in 1 second.

uint8\_t time100ms;

/// System time in seconds.

uint8\_t timeSeconds;

/// System time in minutes.

uint8\_t timeMinutes;

/// System time in hours.

uint8\_t timeHours;

/// Desired LED duty cycle 0 to 9 where 0 = 0% and 9 = 90%.

uint8\_t timeDutyCycle;

/// Current value of the timer 1 compare register used to generate 104uS interrupt rate (9600bps).

uint16\_t timeCompare;

/// 16-bit NCO where the upper 8-bits are used to index into the frequency generation table.

uint16\_t timeNCO;

/// Audio tone NCO update step (phase).

uint16\_t timeNCOFreq;

/// Counter used to deciminate down from the 104uS to 833uS interrupt rate. (9600 to 1200 baud)

uint8\_t timeLowRateCount;

/// Current TNC mode (standby, 1200bps A-FSK, or 9600bps FSK)

TNC\_DATA\_MODE tncDataMode;

/// Flag set true once per second.

bool\_t timeUpdateFlag;

/// Flag that indicate the flight time should run.

bool\_t timeRunFlag;

/// The change in the CCP\_1 register for each 104uS (9600bps) interrupt period.

#define TIME\_RATE 125

/\*\*

\* Running 8-bit counter that ticks every 100mS.

\*

\* @return 100mS time tick

\*/

uint8\_t timeGetTicks()

{

return timeTicks;

}

/\*\*

\* Initialize the real-time clock.

\*/

void timeInit()

{

timeTicks = 0;

timeInterruptCount = 0;

time100mS = 0;

timeSeconds = 0;

timeMinutes = 0;

timeHours = 0;

timeDutyCycle = TIME\_DUTYCYCLE\_70;

timeCompare = TIME\_RATE;

timeUpdateFlag = false;

timeNCO = 0x00;

timeLowRateCount = 0;

timeNCOFreq = 0x2000;

tncDataMode = TNC\_MODE\_STANDBY;

timeRunFlag = false;

// Configure CCP1 to interrupt at 1mS for PSK31 or 833uS for 1200 baud APRS

CCP\_1 = TIME\_RATE;

set\_timer1(timeCompare);

setup\_ccp1( CCP\_COMPARE\_INT );

setup\_timer\_1( T1\_INTERNAL | T1\_DIV\_BY\_4 );

}

/\*\*

\* Function return true once a second based on real-time clock.

\*

\* @return true on one second tick; otherwise false

\*/

bool\_t timeIsUpdate()

{

if (timeUpdateFlag)

{

timeUpdateFlag = false;

return true;

} // END if

return false;

}

/\*\*

\* Set the blink duty cycle of the heartbeat LED. The LED blinks at a 1Hz rate.

\*

\* @param dutyCycle TIME\_DUTYCYCLE\_xx constant

\*/

void timeSetDutyCycle (uint8\_t dutyCycle)

{

timeDutyCycle = dutyCycle;

}

/\*\*

\* Set a flag to indicate the flight time should run. This flag is typically set when the payload

\* lifts off.

\*/

void timeSetRunFlag()

{

timeRunFlag = true;

}

#INT\_CCP1

/\*\*

\* Timer interrupt handler called every 104uS (9600 times/second).

\*/

void timeUpdate()

{

// Setup the next interrupt for the operational mode.

timeCompare += TIME\_RATE;

CCP\_1 = timeCompare;

switch (tncDataMode)

{

case TNC\_MODE\_STANDBY:

break;

case TNC\_MODE\_1200\_AFSK:

ddsSetFTW (freqTable[timeNCO >> 8]);

timeNCO += timeNCOFreq;

if (++timeLowRateCount == 8)

{

timeLowRateCount = 0;

tnc1200TimerTick();

} // END if

break;

case TNC\_MODE\_9600\_FSK:

tnc9600TimerTick();

break;

} // END switch

// Read the GPS serial port and save any incoming characters.

serialUpdate();

// Count the number of milliseconds required for the tenth second counter.

if (++timeInterruptCount == 960)

{

timeInterruptCount = 0;

// This timer just ticks every 100mS and is used for general timing.

++timeTicks;

// Roll the counter over every second.

if (++time100mS == 10)

{

time100mS = 0;

// We set this flag true every second.

timeUpdateFlag = true;

// Maintain a Real Time Clock.

if (timeRunFlag)

if (++timeSeconds == 60)

{

timeSeconds = 0;

if (++timeMinutes == 60)

{

timeMinutes = 0;

++timeHours;

} // END if timeMinutes

} // END if timeSeconds

} // END if time100mS

// Flash the status LED at timeDutyCycle % per second. We use the duty cycle for mode feedback.

if (time100mS >= timeDutyCycle)

output\_low (IO\_LED);

else

output\_high (IO\_LED);

} // END if

}

/\*\* @} \*/

/\*\*

\* @defgroup tnc TNC (Terminal Node Controller)

\*

\* Functions that provide a subset of the TNC functions.

\*

\* @{

\*/

/// The number of start flag bytes to send before the packet message. (360bits \* 1200bps = 300mS)

#define TNC\_TX\_DELAY 45

/// The size of the TNC output buffer.

#define TNC\_BUFFER\_SIZE 80

/// States that define the current mode of the 1200 bps (A-FSK) state machine.

enum TNC\_TX\_1200BPS\_STATE

{

/// Stand by state ready to accept new message.

TNC\_TX\_READY,

/// 0x7E bit stream pattern used to define start of APRS message.

TNC\_TX\_SYNC,

/// Transmit the AX.25 header that contains the source/destination call signs, APRS path, and flags.

TNC\_TX\_HEADER,

/// Transmit the message data.

TNC\_TX\_DATA,

/// Transmit the end flag sequence.

TNC\_TX\_END

};

/// Enumeration of the messages we can transmit.

enum TNC\_MESSAGE\_TYPE

{

/// Startup message that contains software version information.

TNC\_BOOT\_MESSAGE,

/// Plain text status message.

TNC\_STATUS,

/// Message that contains GPS NMEA-0183 $GPGGA message.

TNC\_GGA,

/// Message that contains GPS NMEA-0183 $GPRMC message.

TNC\_RMC

};

/// AX.25 compliant packet header that contains destination, station call sign, and path.

/// 0x76 for SSID-11, 0x78 for SSID-12

uint8\_t TNC\_AX25\_HEADER[30] = {

'A' << 1, 'P' << 1, 'R' << 1, 'S' << 1, ' ' << 1, ' ' << 1, 0x60, \

'K' << 1, 'D' << 1, '7' << 1, 'L' << 1, 'M' << 1, 'O' << 1, 0x76, \

'G' << 1, 'A' << 1, 'T' << 1, 'E' << 1, ' ' << 1, ' ' << 1, 0x60, \

'W' << 1, 'I' << 1, 'D' << 1, 'E' << 1, '3' << 1, ' ' << 1, 0x67, \

0x03, 0xf0 };

/// The next bit to transmit.

uint8\_t tncTxBit;

/// Current mode of the 1200 bps state machine.

TNC\_TX\_1200BPS\_STATE tncMode;

/// Counter for each bit (0 - 7) that we are going to transmit.

uint8\_t tncBitCount;

/// A shift register that holds the data byte as we bit shift it for transmit.

uint8\_t tncShift;

/// Index into the APRS header and data array for each byte as we transmit it.

uint8\_t tncIndex;

/// The number of bytes in the message portion of the AX.25 message.

uint8\_t tncLength;

/// A copy of the last 5 bits we've transmitted to determine if we need to bit stuff on the next bit.

uint8\_t tncBitStuff;

/// Pointer to TNC buffer as we save each byte during message preparation.

uint8\_t \*tncBufferPnt;

/// The type of message to tranmit in the next packet.

TNC\_MESSAGE\_TYPE tncPacketType;

/// Buffer to hold the message portion of the AX.25 packet as we prepare it.

uint8\_t tncBuffer[TNC\_BUFFER\_SIZE];

/// Flag that indicates we want to transmit every 5 seconds.

bool\_t tncHighRateFlag;

/\*\*

\* Initialize the TNC internal variables.

\*/

void tncInit()

{

tncTxBit = 0;

tncMode = TNC\_TX\_READY;

tncPacketType = TNC\_BOOT\_MESSAGE;

tncHighRateFlag = false;

}

/\*\*

\* Determine if the hardware if ready to transmit a 1200 baud packet.

\*

\* @return true if ready; otherwise false

\*/

bool\_t tncIsFree()

{

if (tncMode == TNC\_TX\_READY)

return true;

return false;

}

void tncHighRate(bool\_t state)

{

tncHighRateFlag = state;

}

/\*\*

\* Configure the TNC for the desired data mode.

\*

\* @param dataMode enumerated type that specifies 1200bps A-FSK or 9600bps FSK

\*/

void tncSetMode(TNC\_DATA\_MODE dataMode)

{

switch (dataMode)

{

case TNC\_MODE\_1200\_AFSK:

ddsSetMode (DDS\_MODE\_AFSK);

break;

case TNC\_MODE\_9600\_FSK:

ddsSetMode (DDS\_MODE\_FSK);

// FSK tones at 445.947 and 445.953 MHz

ddsSetFSKFreq (955382980, 955453621);

break;

} // END switch

tncDataMode = dataMode;

}

/\*\*

\* Determine if the seconds value **timeSeconds** is a valid time slot to transmit

\* a message. Time seconds is in UTC.

\*

\* @param timeSeconds UTC time in seconds

\*

\* @return true if valid time slot; otherwise false

\*/

bool\_t tncIsTimeSlot (uint8\_t timeSeconds)

{

if (tncHighRateFlag)

{

if ((timeSeconds % 5) == 0)

return true;

return false;

} // END if

switch (timeSeconds)

{

case 0:

case 15:

case 30:

case 45:

return true;

default:

return false;

} // END switch

}

/\*\*

\* Method that is called every 833uS to transmit the 1200bps A-FSK data stream.

\* The provides the pre and postamble as well as the bit stuffed data stream.

\*/

void tnc1200TimerTick()

{

// Set the A-FSK frequency.

if (tncTxBit == 0x00)

timeNCOFreq = 0x2000;

else

timeNCOFreq = 0x3aab;

switch (tncMode)

{

case TNC\_TX\_READY:

// Generate a test signal alteranting between high and low tones.

tncTxBit = (tncTxBit == 0 ? 1 : 0);

break;

case TNC\_TX\_SYNC:

// The variable tncShift contains the lastest data byte.

// NRZI enocde the data stream.

if ((tncShift & 0x01) == 0x00)

if (tncTxBit == 0)

tncTxBit = 1;

else

tncTxBit = 0;

// When the flag is done, determine if we need to send more or data.

if (++tncBitCount == 8)

{

tncBitCount = 0;

tncShift = 0x7e;

// Once we transmit x mS of flags, send the data.

// txDelay bytes \* 8 bits/byte \* 833uS/bit = x mS

if (++tncIndex == TNC\_TX\_DELAY)

{

tncIndex = 0;

tncShift = TNC\_AX25\_HEADER[0];

tncBitStuff = 0;

tncMode = TNC\_TX\_HEADER;

} // END if

} else

tncShift = tncShift >> 1;

break;

case TNC\_TX\_HEADER:

// Determine if we have sent 5 ones in a row, if we have send a zero.

if (tncBitStuff == 0x1f)

{

if (tncTxBit == 0)

tncTxBit = 1;

else

tncTxBit = 0;

tncBitStuff = 0x00;

return;

} // END if

// The variable tncShift contains the lastest data byte.

// NRZI enocde the data stream.

if ((tncShift & 0x01) == 0x00)

if (tncTxBit == 0)

tncTxBit = 1;

else

tncTxBit = 0;

// Save the data stream so we can determine if bit stuffing is

// required on the next bit time.

tncBitStuff = ((tncBitStuff << 1) | (tncShift & 0x01)) & 0x1f;

// If all the bits were shifted, get the next byte.

if (++tncBitCount == 8)

{

tncBitCount = 0;

// After the header is sent, then send the data.

if (++tncIndex == sizeof(TNC\_AX25\_HEADER))

{

tncIndex = 0;

tncShift = tncBuffer[0];

tncMode = TNC\_TX\_DATA;

} else

tncShift = TNC\_AX25\_HEADER[tncIndex];

} else

tncShift = tncShift >> 1;

break;

case TNC\_TX\_DATA:

// Determine if we have sent 5 ones in a row, if we have send a zero.

if (tncBitStuff == 0x1f)

{

if (tncTxBit == 0)

tncTxBit = 1;

else

tncTxBit = 0;

tncBitStuff = 0x00;

return;

} // END if

// The variable tncShift contains the lastest data byte.

// NRZI enocde the data stream.

if ((tncShift & 0x01) == 0x00)

if (tncTxBit == 0)

tncTxBit = 1;

else

tncTxBit = 0;

// Save the data stream so we can determine if bit stuffing is

// required on the next bit time.

tncBitStuff = ((tncBitStuff << 1) | (tncShift & 0x01)) & 0x1f;

// If all the bits were shifted, get the next byte.

if (++tncBitCount == 8)

{

tncBitCount = 0;

// If everything was sent, transmit closing flags.

if (++tncIndex == tncLength)

{

tncIndex = 0;

tncShift = 0x7e;

tncMode = TNC\_TX\_END;

} else

tncShift = tncBuffer[tncIndex];

} else

tncShift = tncShift >> 1;

break;

case TNC\_TX\_END:

// The variable tncShift contains the lastest data byte.

// NRZI enocde the data stream.

if ((tncShift & 0x01) == 0x00)

if (tncTxBit == 0)

tncTxBit = 1;

else

tncTxBit = 0;

// If all the bits were shifted, get the next one.

if (++tncBitCount == 8)

{

tncBitCount = 0;

tncShift = 0x7e;

// Transmit two closing flags.

if (++tncIndex == 2)

{

tncMode = TNC\_TX\_READY;

// Tell the TNC time interrupt to stop generating the frequency words.

tncDataMode = TNC\_MODE\_STANDBY;

// Key off the DDS.

output\_low (IO\_OSK);

output\_low (IO\_PTT);

ddsSetMode (DDS\_MODE\_POWERDOWN);

return;

} // END if

} else

tncShift = tncShift >> 1;

break;

} // END switch

}

/\*\*

\* Method that is called every 104uS to transmit the 9600bps FSK data stream.

\*/

void tnc9600TimerTick()

{

}

/\*\*

\* Write **character** to the TNC buffer. Maintain the pointer

\* and length to the buffer. The pointer tncBufferPnt and tncLength

\* must be set before calling this function for the first time.

\*

\* @param character to save to telemetry buffer

\*/

void tncTxByte (uint8\_t character)

{

\*tncBufferPnt++ = character;

++tncLength;

}

/\*\*

\* Generate the GPS NMEA standard UTC time stamp. Data is written through the tncTxByte

\* callback function.

\*/

void tncNMEATime()

{

// UTC of position fix.

printf (tncTxByte, "%02d%02d%02d,", gpsPosition.hours, gpsPosition.minutes, gpsPosition.seconds);

}

/\*\*

\* Generate the GPS NMEA standard latitude/longitude fix. Data is written through the tncTxByte

\* callback function.

\*/

void tncNMEAFix()

{

uint8\_t dirChar;

uint32\_t coord, coordMin;

// Latitude value.

coord = gpsPosition.latitude;

if (gpsPosition.latitude < 0)

{

coord = gpsPosition.latitude \* -1;

dirChar = 'S';

} else {

coord = gpsPosition.latitude;

dirChar = 'N';

}

coordMin = (coord % 3600000) / 6;

printf (tncTxByte, "%02ld%02ld.%04ld,%c,", (uint32\_t) (coord / 3600000), (uint32\_t) (coordMin / 10000), (uint32\_t) (coordMin % 10000), dirChar);

// Longitude value.

if (gpsPosition.longitude < 0)

{

coord = gpsPosition.longitude \* - 1;

dirChar = 'W';

} else {

coord = gpsPosition.longitude;

dirChar = 'E';

}

coordMin = (coord % 3600000) / 6;

printf (tncTxByte, "%03ld%02ld.%04ld,%c,", (uint32\_t) (coord / 3600000), (uint32\_t) (coordMin / 10000), (uint32\_t) (coordMin % 10000), dirChar);

}

/\*\*

\* Generate the GPS NMEA-0183 $GPGGA packet. Data is written through the tncTxByte

\* callback function.

\*/

void tncGPGGAPacket()

{

// Generate the GPGGA message.

printf (tncTxByte, "$GPGGA,");

// Standard NMEA time.

tncNMEATime();

// Standard NMEA-0183 latitude/longitude.

tncNMEAFix();

// GPS status where 0: not available, 1: available

if (gpsGetFixType() != GPS\_NO\_FIX)

printf (tncTxByte, "1,");

else

printf (tncTxByte, "0,");

// Number of visible birds.

printf (tncTxByte, "%02d,", gpsPosition.trackedSats);

// DOP

printf (tncTxByte, "%ld.%01ld,", gpsPosition.dop / 10, gpsPosition.dop % 10);

// Altitude in meters.

printf (tncTxByte, "%ld.%02ld,M,,M,,", (int32\_t) (gpsPosition.altitudeCM / 100l), (int32\_t) (gpsPosition.altitudeCM % 100));

// Checksum, we add 1 to skip over the $ character.

printf (tncTxByte, "\*%02X", gpsNMEAChecksum(tncBuffer + 1, tncLength - 1));

}

/\*\*

\* Generate the GPS NMEA-0183 $GPRMC packet. Data is written through the tncTxByte

\* callback function.

\*/

void tncGPRMCPacket()

{

uint32\_t temp;

// Generate the GPRMC message.

printf (tncTxByte, "$GPRMC,");

// Standard NMEA time.

tncNMEATime();

// GPS status.

if (gpsGetFixType() != GPS\_NO\_FIX)

printf (tncTxByte, "A,");

else

printf (tncTxByte, "V,");

// Standard NMEA-0183 latitude/longitude.

tncNMEAFix();

// Speed knots and heading.

temp = (int32\_t) gpsPosition.hSpeed \* 75000 / 385826;

printf (tncTxByte, "%ld.%ld,%ld.%ld,", (int16\_t) (temp / 10), (int16\_t) (temp % 10), gpsPosition.heading / 10, gpsPosition.heading % 10);

// Date

printf (tncTxByte, "%02d%02d%02ld,,", gpsPosition.day, gpsPosition.month, gpsPosition.year % 100);

// Checksum, skip over the $ character.

printf (tncTxByte, "\*%02X", gpsNMEAChecksum(tncBuffer + 1, tncLength - 1));

}

/\*\*

\* Generate the plain text status packet. Data is written through the tncTxByte

\* callback function.

\*/

void tncStatusPacket(int16\_t temperature)

{

uint16\_t voltage;

// Plain text telemetry.

printf (tncTxByte, ">ANSR ");

// Display the flight time.

printf (tncTxByte, "%02U:%02U:%02U ", timeHours, timeMinutes, timeSeconds);

// Altitude in feet.

printf (tncTxByte, "%ld' ", gpsPosition.altitudeFeet);

// Peak altitude in feet.

printf (tncTxByte, "%ld'pk ", gpsGetPeakAltitude());

// GPS hdop or pdop

printf (tncTxByte, "%lu.%lu", gpsPosition.dop / 10, gpsPosition.dop % 10);

// The text 'pdop' for a 3D fix, 'hdop' for a 2D fix, and 'dop' for no fix.

switch (gpsGetFixType())

{

case GPS\_NO\_FIX:

printf (tncTxByte, "dop ");

break;

case GPS\_2D\_FIX:

printf (tncTxByte, "hdop ");

break;

case GPS\_3D\_FIX:

printf (tncTxByte, "pdop ");

break;

} // END switch

// Number of satellites in the solution.

printf (tncTxByte, "%utrk ", gpsPosition.trackedSats);

// Display main bus voltage.

voltage = adcGetMainBusVolt();

printf (tncTxByte, "%lu.%02luvdc ", voltage / 100, voltage % 100);

// Display internal temperature.

printf (tncTxByte, "%ld.%01ldF ", temperature / 10, abs(temperature % 10));

// Print web address link.

printf (tncTxByte, "www.kd7lmo.net");

}

/\*\*

\* Prepare an AX.25 data packet. Each time this method is called, it automatically

\* rotates through 1 of 3 messages.

\*

\* @param dataMode enumerated type that specifies 1200bps A-FSK or 9600bps FSK

\*/

void tncTxPacket(TNC\_DATA\_MODE dataMode)

{

int16\_t temperature;

uint16\_t crc;

// Only transmit if there is not another message in progress.

if (tncMode != TNC\_TX\_READY)

return;

// Log the battery and reference voltage before we start the RF chain.

sysLogVoltage();

// We need to read the temperature sensor before we setup the DDS since they share a common clock pin.

temperature = lm92GetTemp();

// Log the system temperature every time we transmit a packet.

logType (LOG\_TEMPERATURE);

logInt16 (temperature);

// Configure the DDS for the desired operational.

tncSetMode (dataMode);

// Set a pointer to our TNC output buffer.

tncBufferPnt = tncBuffer;

// Set the message length counter.

tncLength = 0;

// Determine the contents of the packet.

switch (tncPacketType)

{

case TNC\_BOOT\_MESSAGE:

printf (tncTxByte, ">ANSR Pico Beacon - V3.05");

// Select the next packet we will generate.

tncPacketType = TNC\_STATUS;

break;

case TNC\_STATUS:

tncStatusPacket(temperature);

// Select the next packet we will generate.

tncPacketType = TNC\_GGA;

break;

case TNC\_GGA:

tncGPGGAPacket();

// Select the next packet we will generate.

tncPacketType = TNC\_RMC;

break;

case TNC\_RMC:

tncGPRMCPacket();

// Select the next packet we will generate.

tncPacketType = TNC\_STATUS;

break;

}

// Add the end of message character.

printf (tncTxByte, "\015");

// Calculate the CRC for the header and message.

crc = sysCRC16(TNC\_AX25\_HEADER, sizeof(TNC\_AX25\_HEADER), 0xffff);

crc = sysCRC16(tncBuffer, tncLength, crc ^ 0xffff);

// Save the CRC in the message.

\*tncBufferPnt++ = crc & 0xff;

\*tncBufferPnt = (crc >> 8) & 0xff;

// Update the length to include the CRC bytes.

tncLength += 2;

// Prepare the variables that are used in the real-time clock interrupt.

tncBitCount = 0;

tncShift = 0x7e;

tncTxBit = 0;

tncIndex = 0;

tncMode = TNC\_TX\_SYNC;

// Turn on the PA chain.

output\_high (IO\_PTT);

// Wait for the PA chain to power up.

delay\_ms (10);

// Key the DDS.

output\_high (IO\_OSK);

// Log the battery and reference voltage just after we key the transmitter.

sysLogVoltage();

}

/\*\* @} \*/

uint32\_t counter;

uint8\_t bitIndex;

uint8\_t streamIndex;

uint8\_t value;

uint8\_t bitStream[] = { 0x10, 0x20, 0x30 };

void init()

{

counter = 0;

bitIndex = 0;

streamIndex = 0;

value = bitStream[0];

}

void test()

{

counter += 0x10622d;

CCP\_1 = (uint16\_t) ((counter >> 16) & 0xffff);

if ((value & 0x80) == 0x80)

setup\_ccp1 (CCP\_COMPARE\_SET\_ON\_MATCH);

else

setup\_ccp1 (CCP\_COMPARE\_CLR\_ON\_MATCH);

if (++bitIndex == 8)

{

bitIndex = 0;

if (++streamIndex == sizeof(bitStream))

{

streamIndex = 0;

}

value = bitStream[streamIndex];

} else

value = value << 1;

}

// This is where we go after reset.

void main()

{

uint8\_t i, utcSeconds, lockLostCounter;

test();

// Configure the basic systems.

sysInit();

// Wait for the power converter chains to stabilize.

delay\_ms (100);

// Setup the subsystems.

adcInit();

flashInit();

gpsInit();

logInit();

timeInit();

serialInit();

tncInit();

// Program the DDS.

ddsInit();

// Turn off the LED after everything is configured.

output\_low (IO\_LED);

// Check for the diagnostics plug, otherwise we'll continue to boot.

diagPort();

// Setup our interrupts.

enable\_interrupts(GLOBAL);

enable\_interrupts(INT\_CCP1);

// Turn on the GPS engine.

gpsPowerOn();

// Allow the GPS engine to boot.

delay\_ms (250);

// Initialize the GPS engine.

while (!gpsSetup());

// Charge the ADC filters.

for (i = 0; i < 32; ++i)

adcUpdate();

// Log startup event.

logType (LOG\_BOOTED);

logUint8 (gpsPosition.month);

logUint8 (gpsPosition.day);

logUint8 (gpsPosition.year & 0xff);

logUint8 (gpsPosition.hours);

logUint8 (gpsPosition.minutes);

logUint8 (gpsPosition.seconds);

// Transmit software version packet on start up.

tncTxPacket(TNC\_MODE\_1200\_AFSK);

// Counters to send packets if the GPS time stamp is not available.

lockLostCounter = 5;

utcSeconds = 55;

// This is the main loop that process GPS data and waits for the once per second timer tick.

for (;;)

{

// Read the GPS engine serial port FIFO and process the GPS data.

gpsUpdate();

if (gpsIsReady())

{

// Start the flight timer when we get a valid 3D fix.

if (gpsGetFixType() == GPS\_3D\_FIX)

timeSetRunFlag();

// Generate our packets based on the GPS time.

if (tncIsTimeSlot(gpsPosition.seconds))

tncTxPacket(TNC\_MODE\_1200\_AFSK);

// Sync the internal clock to GPS UTC time.

utcSeconds = gpsPosition.seconds;

// This counter is reset every time we receive the GPS message.

lockLostCounter = 0;

// Log the data to flash.

sysLogGPSData();

} // END if gpsIsReady

// Processing that occurs once a second.

if (timeIsUpdate())

{

// We maintain the UTC time in seconds if we shut off the GPS engine or it fails.

if (++utcSeconds == 60)

utcSeconds = 0;

// If we loose information for more than 5 seconds,

// we will determine when to send a packet based on internal time.

if (lockLostCounter == 5)

{

if (tncIsTimeSlot(utcSeconds))

tncTxPacket(TNC\_MODE\_1200\_AFSK);

} else

++lockLostCounter;

// Update the ADC filters.

adcUpdate();

if (timeHours == 5 && timeMinutes == 0 && timeSeconds == 0)

gpsPowerOff();

} // END if timeIsUpdate

} // END for

}