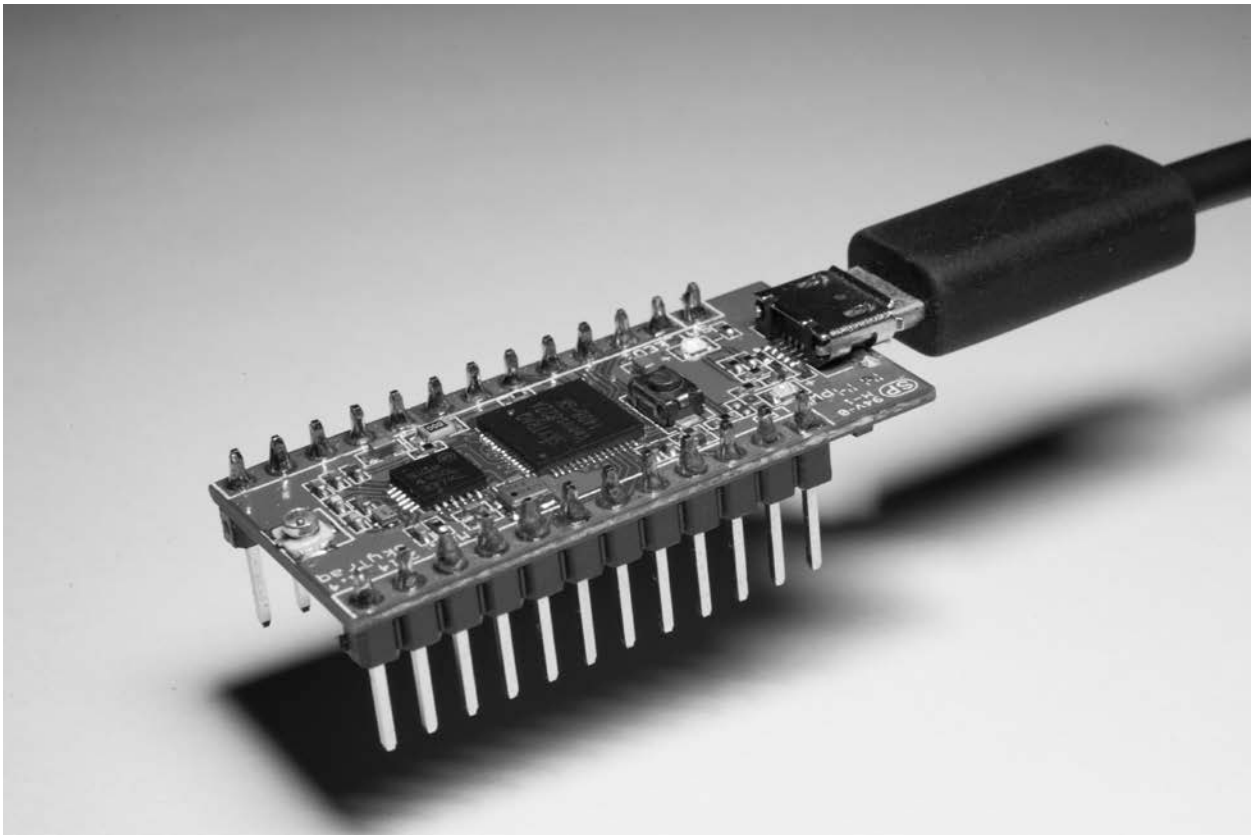
NavSpark



NavSpark Programmer Guide

Rev. 0.4

December 14, 2015

Er

Table of Contents

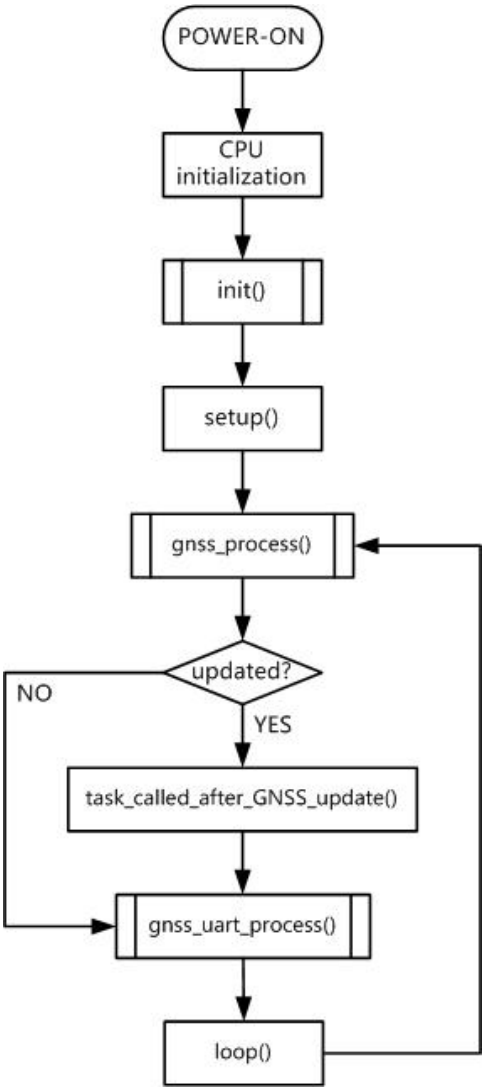
1. Introduction

This manual is intended for users who wish to build application based on the NavSpark development platform using the Arduino IDE. This manual provides an introduction on how NavSpark works and available APIs. As both GNSS mode (LEON3 with GNSS Library) and MCU mode (LEON3 without GNSS Library) are included in the board manager, MCU mode usage is also described.

2. Programmer Guide

2.1 Overview

Below chart provides an idea about the procedure flow of program running on NavSpark.



- After power on, NavSpark will do necessary initializations for the 32-bit LEON3 core, and arrange all software resources including stack/heap, etc.

- Then NavSpark performs further initializations in the “init()” which is defined in “wiring.c”, currently NavSpark initializes the ISR for UART/2-wire/SPI here. It is possible for users to add their code for any other system level initialization in this function, but need to do it with care.

- The next step for NavSpark is to execute the user-defined “setup()” which is auto-generated in sketch by Arduino IDE. The “GnssConf.init()” must be called at here for NavSpark and users may add their code here to setup all necessary tasks for later application. After leaving “setup()”, NavSpark will enter an endless loop

- “gnss_process()” is divided into two parts, GNSS mode and MCU mode. NavSpark will update the GNSS information at certain frequency which can be change by “GNSSParam::setUpdateRate()” and return TRUE in case update is done this time. The default update frequency is 1Hz and can be changed if users choose to link the GNSS library, the “gnss_process()” maintains returning TRUE every 1 second if users choose to link MCU mode.

- There is one function “task_called_after_GNSS_update()” which is defined in “main.cpp” will be executed in case of GNSS update is done. Users may provide their own “task_called_after_GNSS_update()” in sketch to replace the default one. Normally this function is used to process the most recent GNSS navigation result obtained from calling “gnss_process()”.

- “gnss_uart_process()” is divided into two parts, GNSS mode and MCU mode. NavSpark handles the NMEA messages and prints them out via UART1, it also process the binary commands received over UART1. UART1 is connected to micro USB through a UART-to-USB bridge chip. In MCU mode, UART1 is open for users to use.

- The last function in loop is “loop()” which is auto-generated in sketch by Arduino IDE. It is the place for users to add their normal routines in “loop()”. After leaving “loop()”, NavSpark jumps to “gnss_process()” and new cycle begins.

2.2 Using the NavSpark API

Basically NavSpark is an Arduino-based platform and most of the APIs are packaged as member functions of C++ class. To use those member functions, the associated class objects must be instantiated prior to using them. For the functions written in C users can use them directly.

3. NavSpark API Reference

3.1 Data Type

U08	unsigned char (8-bit unsigned integer)
BYTE	
UCHAR	
uint8_t	
S08	signed char (8-bit signed integer)
CHAR	
U16	unsigned short / unsigned short int (16-bit unsigned integer)
USHORT	
WORD	
WCHAR	
uint16_t	
S16	signed short / short (16-bit signed integer)
SHORT	
U32	unsigned int / unsigned long/ unsigned long int (32-bit unsigned integer)
UINT	
ULONG	
DWORD	
size_t	
uint32_t	
S32	long / signed long int (32-bit signed interger)
INT	
LONG	
F32	float (32-bit float precision)
D64	double (64-bit double precision)
U64	unsigned long long int (64-bit unsigned long integer)
S64	Signed long long int (64-bit signed long integer)

For more detail information, please see “stdint.h” and “st_type.h”.

Base	Example	Formatter	Comment
10 (decimal)	123	none	
2 (binary)	B1111011	leading 'B'	only works with 8 bit values (0 to 255)
8 (octal)	0173	leading "0"	characters 0-7 valid
16 (hexadecimal)	0x7B	leading "0x"	characters 0-9, A-F, a-f valid

LEON3 BCC can’t support binary constant.

3.2 API List

- 3.3.1 GNSSParam
- 3.3.2 GNSS
- 3.3.3 GNSSDate
- 3.3.4 GNSSTime
- 3.3.5 GNSSLocation
- 3.3.6 GNSSAltitude
- 3.3.7 GNSSGeoSeparation
- 3.3.8 GNSSDOP
- 3.3.9 GNSSSpeed
- 3.3.10 GNSSCourse
- 3.3.11 GNSSSatellites
- 3.3.12 GNSSTimeStamp
- 3.3.13 Timer
- 3.3.14 SPI_MasterSlave
- 3.3.15 SPIClass
- 3.3.16 TwoWire_MasterSlave
- 3.3.17 TwoWire
- 3.3.18 HardwareSerial
- 3.3.19 Analog
- 3.3.20 Digital
- 3.3.21 SDClass
- 3.3.22 File

3.3 Descriptions to API

3.3.1 GNSSParam

GNSSParam

Class for the default values settings of GNSS receiver.

Public Member Functions

- void setDefault(void)
- void setNavMode(uint8_t mode)
- void setUpdateRate(uint8_t rate)
- void setDopMaskMode(uint8_t mode)
- void setPdopMask(float pdop)
- void setHdopMask(float hdop)
- void setGdopMask(float gdop)
- void init(void)
- bool init_done(void)

Remarks

GnssConf, **GNSSParam** class, is pre-instantiated in “GNSS.cpp”. User can use this object to configure some parameters of the GNSS receiver.

GNSSParam::setDefault()

All default values of GNSS receiver are set by this function.

Syntax

#include “GNSS.h”
GnssConf.setDefault();

Parameters

None

Returns

None

Remarks

This function is already called in the constructor of GNSSParam and it is unnecessary for user to call it manually.

GNSSParam::setNavMode()

It allows user to select navigation mode.

Syntax

#include “GNSS.h”
GnssConf.setNavMode(mode);

Parameters

uint8_t mode

The allowed values arelisted below.

STGNSS_NAV_MODE_AUTO	(default)
STGNSS_NAV_MODE_PEDESTRIAN	
STGNSS_NAV_MODE_CAR	
STGNSS_NAV_MODE_MARINE	
STGNSS_NAV_MODE_BALLOON	
STGNSS_NAV_MODE_AIRBORNE	

Returns

None

Remarks

This function won’t change the GNSS receiver setting until init() is executed.

Unless specifically knowing which application scenario applies, otherwise use AUTO mode for default.

Airborne mode can be used for high-dynamics application, not necessarily for airborne use. Little filtering done and has the least lag among all modes. If it’s used in signal fluctuating non-open sky environments, it might result in poor performance. It’s useful mainly under open sky.

GNSSParam::setDopMaskMode()

It allows selection of the DOP (Dilution of Precision) mask criteria for GNSS receiver.

Syntax

#include “GNSS.h”
GnssConf.setDopMaskMode(mode);

Parameters

uint8_tmode

TheDOP mode for GNSS receiver. The allowedvalues arelisted in below.

STGNSS_DOP_MASK_DISABLE
STGNSS_DOP_MASK_AUTO (default)
STGNSS_DOP_MASK_PDOP
STGNSS_DOP_MASK_HDOP
STGNSS_DOP_MASK_GDOP

Returns

None

Remarks

- STGNSS_DOP_MASK_DISABLE：GNSS receiver will ignore DOP mask check and always give valid fix in case of enough satellites are used for fix.
- STGNSS_DOP_MASK_AUTO：GNSS receiver will apply PDOP mask for 3-D fix and HDOP mask for 2-D fix.
- STGNSS_DOP_MASK_PDOP：GNSS receiver will apply PDOP mask only.
- STGNSS_DOP_MASK_HDOP：GNSS receiver will apply HDOP mask only.
- STGNSS_DOP_MASK_GDOP：GNSS receiver will apply GDOP mask only.

This function won’t change the GNSS receiver DOP mask setting immediately until init() is executed. For given selected DOP mask criteria, valid position fix will be generated when the corresponding DOP value computed is below the selected DOP mask threshold; otherwise no-fix flag will be generated.

2D-fix is when getting navigation solution using 3 satellites signals.

3D-fix is when getting navigation solution using 4 or more satellites signals.

GNSSParam::setPdopMask()

It allows user to set the PDOP (Position DOP) mask value for GNSS receiver.

Syntax

```
#include “GNSS.h”
GnssConf.setPdopMask(value);          default 30.0
```

Parameters

float value

The PDOP mask value for GNSS receiver. The range of allowed values are from 0.5 to 30.0 in 0.1 step, the PDOP mask setting of GNSS receiver won’t be changed in case an invalid value is given.

Returns

None

Remarks

This function won’t change PDOP mask setting of the GNSS receiver immediately until init() is executed**.** If PDOP mask is selected by setDOPMaskMode() and geometry of the satellites used to form navigation solution gives a PDOP value higher than the PDOP mask value, then no-fix flag is generated.

GNSSParam::setHdopMask()

It allows user to set the HDOP (Horizontal DOP) mask value for GNSS receiver.

Syntax

#include “GNSS.h”

GnssConf.setHdopMask(value); default 30.0

Parameters

float value

The HDOP mask value for GNSS receiver. The range of allowed values are from 0.5 to 30.0 in 0.1 step, the HDOP mask setting of GNSS receiver won’t be changed in case an invalid value is given.

Returns

None

Remarks

This function won’t change HDOP mask setting of GNSS receiver immediately until init() is executed**.** If HDOP mask is selected by setDOPMaskMode() and geometry of the satellites used to form navigation solution gives an HDOP value higher than the HDOP mask value, then no-fix flag is generated.

GNSSParam::setGdopMask()

It allows user to set the GDOP (Geometric DOP) mask value for GNSS receiver.

Syntax

#include “GNSS.h”

GnssConf.setGdopMask(value); default 30.0

Parameters

float value

The GDOP mask value for GNSS receiver. The range of allowed values are from 0.5 to 30.0 in 0.1 step, the GDOP mask setting of GNSS receiver won’t be changed in case an invalid value is given.

Returns

None

Remarks

This function won’t change the GDOP setting of GNSS receiver immediately until init() is executed**.** If GDOP mask is selected by setDOPMaskMode() and geometry of the satellites used to form PVT solution gives a GDOP value higher than the GDOP mask value, then no-fix flag is generated.

GNSSParam::init()

It performs the initialization for GNSS receiver with default values or the values changed by user.

Syntax

#include “GNSS.h”
GnssConf.init();

Parameters

None

Returns

None

Remarks

This function does the actual changes to the settings of GNSS receiver and perform all necessary initializations including initialization for gnss_process() used in main.cpp. This function must be called when using both GNSS and MCU library, and it is RECOMMENDED to place this function at the beginning of setup() which is generated in sketch.

GNSSParam::init_done()

It allows user to check if GNSS receiver initialization is done or not.

Syntax

#include “GNSS.h”

bool result = GnssConf.init_done();

Parameters

None

Returns

Boolean TRUE if the initialization for GNSS receiver was done or otherwise FALSE.

Remarks

None

3.2.2 GNSS

GNSS

Class for key functions of GNSS receiver.

Public Member Functions

- GNSS(void)
- void update(void)
- bool isUpdated(void)
- uint8_t fixMode(void)
- static double distanceBetween(double lat1, double lon1, double lat2, double lon2)
- static double courseTo(double lat1, double lon1, double lat2, double lon2)

Public Members

- GNSSDate date
- GNSSTime time
- GNSSLocation location
- GNSSAltitude altitude
- GNSSSpeed speed
- GNSSCourse course
- GNSSSatellites satellites
- GNSSTimeStamp timestamp

Remarks

There is one **GNSS** object, GnssInfo, which is pre-instantiated in “GNSS.cpp” when user selects “Leon3 with GNSS library” in Processor menu. User can use this object to retrieve navigation solution and access related operations.

GNSS::update()

This function extracts navigation solution from GNSS kernel and updates to private space of members.

Syntax

#include “GNSS.h”
GnssInfo.update();

Parameters

None

Returns

None

Remarks

This function can be called at any time to extract navigation solution from GNSS receiver. Normally it is placed inside “task_called_after_GNSS_update()” to have the navigation result ready for application use, such as generating NMEA messages.

GNSS::isUpdated()

This function checks if GNSS receiver information has been updated.

Syntax

```
#include “GNSS.h”
bool result = GnssInfo.isUpdated();
```

Parameters

None

Returns

Boolean TRUE is returned if GNSS receiver has new information, either valid fix or invalid fix; otherwise boolean FALSE is returned.

Remarks

None

GNSS::fixMode()

This function is used to get the fix status of GNSS receiver.

Syntax

```
#include “GNSS.h”
uint8_t result = GnssInfo.fixMode();
```

Parameters

None

Returns

An 8-bit unsigned integer to indicate status of GNSS fix.

- 0 : non-fix.
- 1 : prediction
- 2 : 2D fix
- 3 : 3D fix
- 4 : differential mode

Remarks

-

GNSS::distanceBetween()

This function calculates the distance between two separate locations.

Syntax

#include “GNSS.h”

double distance = GnssInfo.distanceBetween(lat1, lon1, lat2, lon2);

Parameters

double lat1

The 1st argument is location 1 latitude in double precision floating point

double long1

The 2nd argument is location 1 longitude in double precision floating point

double lat2

The 3rd argument is location 2 latitude in double precision floating point

double long2

The 4th argument is location 2 longitude in double precision floating point

Returns

A double-precision floating point value that indicates the distance between the two locations in meters.

Remarks

The input arguments are in degrees.

Since the earth is not ideal sphere, this calculation result from this function may have round error up to 0.5%.

GNSS::courseTo()

This function calculates the course from location 1 to location 2.

Syntax

#include “GNSS.h”

double course = GnssInfo.courseTo(lat1, lon1, lat2, lon2);

Parameters

double lat1

The 1st argument is location 1 latitude in double precision floating point

double long1

The 2nd argument is location 1 longitude in double precision floating point

double lat2

The 3rd argument is location 2 latitude in double precision floating point

double long2

The 4th argument is location 2longitude in double precision floating point

Returns

A double-precision floating point value indicates the course in degrees from location 1 to location 2. Degree 0 means the course is toward north and 270 is toward west.

Remarks

The input arguments are in degrees.

Since the earth is not ideal sphere, this calculation result from this function may be off true direction by a tiny fraction.

3.3.3 GNSSDate

GNSSDate

Class to store date information of GNSS receiver, including years, months and days.

Public Member Functions

- void update(uint16_t year, uint8_t month, uint8_t day)
- uint16_t year(void)
- uint8_t month(void))
- uint8_t day(void)
- uint16_t formatString(char* str)

Remarks

None

GNSSDate::update()

This function copies years, months and days given as arguments to its private internal space.

Syntax

#include “GNSS.h”

GnssInfo.date.update(year, month, day);

Parameters

uint16_t year

The 1st argument is the years in AD. This function will abort in case **year** given is less than 1983.

uint8_t month

The 2nd argument is the months. This function will abort in case **month** given is larger than 12 or equal to zero.

uint8_t day

The 3rd argument is the days. This function will abort in case **day** given is larger than 31 or equal to zero.

Returns

None

Remarks

In class **GNSS**, there is a pre-defined class member, date, which type is GNSSDate and date.update() is called inside GNSS::update() to set the date information obtained from GNSS receiver.

GNSSDate::year()

This function returns the years in AD which was set by GNSSDate::update().

Syntax

#include “GNSS.h”
uint16_t my_year = GnssInfo.date.year();

Parameters

None

Returns

A 16-bit unsigned integer representing the years in AD.

Remarks

None

GNSSDate::month()

This function returns the months which was set by GNSSDate::update().

Syntax

#include “GNSS.h”

uint8_t my_month = GnssInfo.date.month();

Parameters

None

Returns

An 8-bit unsigned integer representing the months, ranges from 1 ~ 12.

Remarks

None

GNSSDate::day()

This function returns the days which was set by GNSSDate::update().

Syntax

#include “GNSS.h”

uint8_t my_day = GnssInfo.date.day();

Parameters

None

Returns

An 8-bit unsigned integer representing the days, ranges from 1 ~ 31.

Remarks

None

3.3.4 GNSSTime

GNSSTime

A Class to store information of GNSS time including hour, minute and second.

Public Member Functions

- void update(uint8_t hour, uint8_t min, float second)
- uint8_t hour(void)
- uint8_t minute(void)
- uint8_t second(void)
- uint8_t centisecond(void)
- uint16_t formatString(char* str)

Remarks

None

GNSSTime::update()

This function copies hours, minutes and seconds given as arguments to its private internal space.

Syntax

#include “GNSS.h”

GnssInfo.time.update(hour, min, sec);

Parameters

uint8_t hour

The 1st argument is the value of hours in 24-hour clock system. This function will abort in case **hour** given is larger than 23.

uint8_t min

The 2nd argument is the value for minutes. This function will abort in case **min** given is larger than 59.

float sec

The 3rd argument is the value for seconds. This function will abort in case **sec** given is not less than 60.

Returns

None

Remarks

In class **GNSS**, there is a pre-defined class member, time, which type is GNSSTime and the time.update() is called inside GNSS::update() to set the time information obtained from GNSS receiver.

GNSSTime::hour()

This function returns the hours which was set by GNSSTime::update().

Syntax

#include “GNSS.h”
uint8_t my_hour = GnssInfo.time.hour();

Parameters

None

Returns

An 8-bit unsigned integer representing the hours, ranges from 0 ~ 23.

Remarks

None

GNSSTime::minute()

This function returns the minutes which was set by GNSSTime::update().

Syntax

```
#include “GNSS.h”
uint8_t my_minute = GnssInfo.time.minute();
```

Parameters

None

Returns

An 8-bit unsigned integer representing the minutes, ranges from 0 ~ 59.

Remarks

None

GNSSTime::second()

This function returns the integer part of seconds which was set by GNSSTime::update().

Syntax

#include “GNSS.h”

uint8_t my_second = GnssInfo.time.second();

Parameters

None

Returns

An 8-bit unsigned integer representing the integer part of seconds,ranging from 0 ~ 59.

Remarks

None

GNSSTime::centisecond()

This function returns the fractional part of seconds which was set by GNSSTime::update().

Syntax

#include “GNSS.h”

uint16_t my_centi_second = GnssInfo.time.centisecond();

Parameters

None

Returns

A 16-bit unsigned integer representing the fractional part of second multiplies by 100, ranges from 0 ~ 99.

Remarks

Following example gives a more clear description for how to use GNSSTime.

```
GnssInfo.time.update(13, 45, 29.145); /* save 13:45:29.145 */
GnssInfo.time.hour(); /* returns 13 */
GnssInfo.time.minute(); /* returns 45 */
GnssInfo.time.second(); /* returns 29 */
GnssInfo.time.centisecond(); /* returns 14 */
```

GNSSTime::formatString()

This function converts the time saved by GNSSTime::update() to a string in a pre-defined format.

Syntax

#include “GNSS.h”

uint16_tsizeOfString = GnssInfo.time.formatString(str);

Parameters

char* str

An address pointer points to a character buffer which will contain the string.

Returns

A 16-bit unsigned integer which indicate the size of string after function call.

Remarks

Following example gives a more clear description for how to use GNSSTime.

```
char str[32];
GnssInfo.time.update(13, 45, 29.145); /* save 13:45:29.145*/
uint16_tsizeOfString = GnssInfo.time.formatString(str);
/* -- now we have
sizeOfString =>11
str =>“01:45:29 PM”
*/
```

3.3.5 GNSSLocation

GNSSLocation

Class to store the information of GNSS position, including latitude and longitude.

Public Member Functions

- void update(double lat, double long)
- double latitude(void)
- double longitude(void)
- uint16_t latitude_formatString(char* str)
- uint16_t longitude_formatString(char* str)

Remarks

None

GNSSLocation::update()

This function copies latitude and longitude given as arguments to its private internal space.

Syntax

#include “GNSS.h”

GnssInfo.location.update(lat, long);

Parameters

double lat

The 1st argument is the latitude in degrees with double precision. The allowed range is from -90 to +90.

double long

The 2nd argument is the longitude in degrees with double precision. The allowed range is from -180 to +180.

Returns

None

Remarks

In class **GNSS**, there is a pre-defined class member, location, which type is **GNSSLocation** and the location.update() is called inside GNSS::update() to set the latitude and longitude obtained from GNSS receiver.

GNSSLocation::latitude()

This function returns the latitude which was set by GNSSLocation::update().

Syntax

#include “GNSS.h”

double my_latitude = GnssInfo.location.latitude();

Parameters

None

Returns

A double-precision float point value for latitude in degrees, within $\pm 90°$.

Remarks

None

GNSSLocation::longitude()

This function returns the longitude which was set by GNSSLocation::update().

Syntax

#include “GNSS.h”
double my_longitude = GnssInfo.location.longitude();

Parameters

None

Returns

A double-precision float point value for longitude in degrees, within $\pm 180°$.

Remarks

None

GNSSLocation::latitude_formatString()

This function converts the latitude saved by GNSSLocation::update() to a string in a pre-defined format.

Syntax

#include “GNSS.h”

uint16_t sizeOfString = GnssInfo.location.latitude_formatString(str);

Parameters

char* str

An address pointer points to a character buffer which will contain the string.

Returns

A 16-bit unsigned integer which indicate the size of string.

Remarks

See remarks of **GNSSLocation::longitude_formatString()** for more info.

3.3.6 GNSSAltitude

GNSSAltitude

A Class to store the information of mean sea level (MSL) altitude.

Public Member Functions

- void update(float altitude)
- float meters(void)
- float miles(void)
- float kilometers(void)
- float feet(void)

Remarks

None

GNSSAltitude::update()

This function copies the altitude given as argument to its private internal space.

Syntax

#include “GNSS.h”
GnssInfo.altitude.update(alt);

Parameters

float alt
A floating point value to specify the altitude in meters.

Returns

None

Remarks

In class **GNSS**, there is a pre-defined class member, altitude, which type is **GNSSAltitude** and the altitude.update() is called inside GNSS::update() to set the mean sea level (MSL) altitude in meters obtained from GNSS receiver.

GNSSAltitude::meters()

This function returns the MSL altitude set by GNSSAltitude::update() in meters.

Syntax

#include “GNSS.h”
float my_altitude = GnssInfo.altitude.meters();

Parameters

None

Returns

A floating point value to indicate the altitude in meters.

Remarks

None

GNSSAltitude::kilometers()

This function returns the MSL altitude set by GNSSAltitude::update() in kilometers.

Syntax

#include “GNSS.h”
float my_altitude = GnssInfo.altitude.kilometers();

Parameters

None

Returns

A floating point value to indicate the altitude in kilometers.

Remarks

None

GNSSAltitude::feet()

This function returns the MSL altitude set by GNSSAltitude::update() in feet.

Syntax

#include “GNSS.h”
float my_altitude = GnssInfo.altitude.feet();

Parameters

None

Returns

A floating point value to indicate the altitude in feet.

Remarks

None

3.3.7 GNSSGeoSeparation

GNSSGeoSeparation

A Class to store the information of geoid separation.

Public Member Functions

- void update(float meters)
- float meters(void)
- float miles(void)
- float kilometers(void)
- float feet(void)

Remarks

None

GNSSGeoSeparation::update()

This function copies the geoid separation given as argument to its private internal space.

Syntax

#include “GNSS.h”

GnssInfo. geoseperation.update(meter);

Parameters

float meter

A floating point value to specify the geoid separation in meters.

Returns

None

Remarks

In class **GNSS**, there is a pre-defined class member, altitude, which type is **GNSSGeoSeparation** and the geoseperation.update() is called inside GNSS::update() to set the geoid separation in meters obtained from GNSS receiver.

GNSSGeoSeparation::meters()

This function returns the geoid separation set by GNSSGeoSeparation::update() in meters.

Syntax

#include “GNSS.h”

float local_separation = GnssInfo.geoseparation.meters();

Parameters

None

Returns

A floating point value to indicate the geoid separation in meters.

Remarks

None

GNSSGeoSeparation::miles()

This function returns the geoid separation set by GNSSGeoSeparation::update() in meters.

Syntax

#include “GNSS.h”

float local_separation = GnssInfo.geoseparation.miles();

Parameters

None

Returns

A floating point value to indicate the geoid separation in miles.

Remarks

None

GNSSGeoSeparation::kilometers()

This function returns the geoid separation set by GNSSGeoSeparation::update() in meters.

Syntax

#include “GNSS.h”

float local_separation = GnssInfo.geoseparation.kilometers();

Parameters

None

Returns

A floating point value to indicate the geoid separation in kilometers.

Remarks

None

GNSSGeoSeparation::feet()

This function returns the geoid separation set by GNSSGeoSeparation::update() in meters.

Syntax

#include “GNSS.h”
float local_separation = GnssInfo.geoseparation.feet();

Parameters

None

Returns

A floating point value to indicate the geoid separation in feet.

Remarks

None

3.3.8 GNSSDOP

GNSSDOP

A Class to store the information of Dilution of Precision (DOP).

Public Member Functions

- float pdop(void)
- float hdop(void)
- float vdop(void)

Remarks

None

GNSSDOP::pdop()

This function returns current Position Dilution of Precision (PDOP).

Syntax

#include “GNSS.h”
float current_pdop = GnssInfo.dop.pdop();

Parameters

None

Returns

A floating point value to indicate current PDOP value.

Remarks

None

GNSSDOP::hdop()

This function returns current Horizontal Dilution of Precision (HDOP).

Syntax

#include “GNSS.h”
float current_hdop = GnssInfo.dop.hdop();

Parameters

None

Returns

A floating point value to indicate current HDOP value.

Remarks

None

GNSSDOP::vdop()

This function returns current Vertical Dilution of Precision (VDOP).

Syntax

#include “GNSS.h”
float current_vdop = GnssInfo.dop.vdop();

Parameters

None

Returns

A floating point value to indicate current VDOP value.

Remarks

None

3.3.9 GNSSSpeed

GNSSSpeed

Class to store the information of speed.

Public Member Functions

- void update(float speed)
- float kph(void)
- float knots(void)
- float mph(void)

Remarks

None

GNSSSpeed::update()

This function copies the speed given as argument to its private internal space.

Syntax

#include “GNSS.h”
GnssInfo.speed.update(speed);

Parameters

floatspeed
A floating point value to specify the speed in knots.

Returns

None

Remarks

In class **GNSS**, there is a pre-defined class member, speed, which type is **GNSSSpeed** and the speed.update() is called inside GNSS::update() to set the speed in knots obtained from GNSS receiver.

GNSSSpeed::kph()

This function returns the speed set by GNSSSpeed::update() in kilometers per hour.

Syntax

```
#include “GNSS.h”
float my_speed = GnssInfo.speed.kph();
```

Parameters

None

Returns

A floating point value to indicate the speed in kilometers per hour.

Remarks

None

GNSSSpeed::mph()

This function returns the speed set by GNSSSpeed::update() in miles per hour.

Syntax

#include “GNSS.h”
float my_speed = GnssInfo.speed.mph();

Parameters

None

Returns

A floating point value to indicate the speed in miles per hour.

Remarks

None

3.3.10 GNSSCourse

GNSSCourse

Class to store the information of course heading.

Public Member Functions

- void update(float course)
- float deg(void)

Remarks

None

GNSSCourse::update()

This function copies the course in degree given as argument to its private internal space.

Syntax

#include “GNSS.h”

GnssInfo.course.update(deg);

Parameters

floatdeg

A floating point value to specify the course in degree. This function only accept degree if $0 \leq \deg < 360$.

Returns

None

Remarks

In class **GNSS**, there is a pre-defined class member, course, which type is **GNSSCourse** and the course.update() is called inside GNSS::update() to set the course in degrees obtained from GNSS receiver**.**

GNSSCourse::deg()

This function returns the course set by GNSSSpeed::update() in degree.

Syntax

#include “GNSS.h”
float my_course = GnssInfo.course.deg();

Parameters

None

Returns

A floating point value to indicate the course in degree.

Remarks

None

3.3.11 GNSSSatellites

GNSSSatellites

Class to store the information of satellites seen by GNSS receiver.

Public Member Functions

- void update(PVT_DATA_T* pPvtData, SV_INFO_T* pSvInfo)
- uint16_t numGPSInView(uint16_t *prn)
- uint16_t numBD2InView(uint16_t *prn)
- uint16_t numGLNInView(uint16_t *prn)
- uint16_t numGPSInUse(uint16_t *prn)
- uint16_t numBD2InUse(uint16_t *prn)
- uint16_t numGLNInUse(uint16_t *prn)
- uint16_t elevation(uint8_t constellation, uint16_t prn)
- uint16_t azimuth(uint8_t constellation, uint16_t prn)
- uint16_t CNR(uint8_t constellation, uint16_t prn)

Remarks

None

GNSSSatellites::numGPSInView()

This function returns the number and PRN list for GPS satellites in view which was set by GNSSSatellites::update().

Syntax

#include “GNSS.h”

uint16_t num = GnssInfo.satellites.numGPSInView(prnList);

Parameters

uint16_t*prnList

An address pointer points to an array of unsigned 16-bit values with number of array elements equal to **STGNSS_GPS_NCHAN** which is defined in “sti_gnss_lib.h”. If this pointer is not NULL then this array will contain PRN listof in view GPS satellites determined by GNSS receiver.If a NULL pointer is given, then no PRN list will be returned.

Returns

A 16-bit unsigned integer indicating number of in view GPS satellites determined by GNSS receiver.

Remarks

The PRN in list may not be in order. Assumes there are totally 4 GPS satellites in view and their PRN are 1/3/5/7, the prnList[0] may be 5 and prnList[1] may be 1, and so on.

GNSSSatellites::numBD2InView()

This function returns the number and PRN listforBeidou2 satellites in view which was set by GNSSSatellites::update().

Syntax

#include “GNSS.h”

uint16_t num = GnssInfo.satellites.numBD2InView(prnList);

Parameters

uint16_t* prnList

An address pointer points to an array of unsigned 16-bit values with number of array elements equal to **STGNSS_BD2_NCHAN** which is defined in “sti_gnss_lib.h”. If this pointer is not NULL then this array will contain the PRN listof in view Beidou2 satellites determined by GNSS receiver. If a NULL pointer is given, then no PRN list will be returned.

Returns

A 16-bit unsigned integer indicating number of in view Beidou2 satellites determined by GNSS receiver.

Remarks

None

GNSSSatellites::numGLNInView()

This function returns the number and PRN list for GLONASS satellites in view which was set by GNSSSatellites::update().

Syntax

#include “GNSS.h”

uint16_t num = GnssInfo.satellites.numGLNInView(prnList);

Parameters

uint16_t*prnList

An address pointer points to an array of unsigned 16-bit values with number of array elementsequal to **STGNSS_GLONASS_NCHAN** which is defined in “sti_gnss_lib.h”. If this pointer is not NULL then this array will contain the PRN list for in view GLONASS satellites determined by GNSS receiver. If a NULL pointer is given, then no PRN list will be returned.

Returns

A 16-bit unsigned integer indicating number of in view GLONASS satellites determined by GNSS receiver.

Remarks

None

GNSSSatellites::numGPSInUse()

This function returns the number of GPS satellites and their PRN list that is used in computing navigation solution.

Syntax

#include “GNSS.h”
uint16_t num = GnssInfo.satellites.numGPSInUse(prnList);

Parameters

uint16_t*prnList

An address pointer points to an array of unsigned 16-bit values with number of array elements equal to **STGNSS_GPS_NCHAN** which is defined in “sti_gnss_lib.h”. If this pointer is not NULL then this array will contain the PRN list for GPS satellites used for GNSS fix by GNSS receiver. If a NULL pointer is given, then no PRN list will be returned.

Returns

A 16-bit unsigned integer to indicate the number of GPS satellites used by GNSS receiver in calculation GNSS fix. The number in use is less than or equal to the number in view.

Remarks

None

GNSSSatellites::numBD2InUse()

This function returns the number and PRN list forBeidou2 satellites used in computing navigation solution.

Syntax

#include “GNSS.h”

uint16_t num = GnssInfo.satellites.numBD2InUse(prnList);

Parameters

uint16_t* prnList

An address pointer points to an array of unsigned 16-bit values with number of array elements equal to **STGNSS_BD2_NCHAN** which is defined in “sti_gnss_lib.h”. If this pointer is not NULL then this array will contain the PRN list for Beidou2 satellites used for GNSS fix by GNSS receiver. If a NULL pointer is given, then no PRN list will be returned.

Returns

A 16-bit unsigned integer to indicate the number of Beidou2 satellites used by GNSS receiver in calculation for GNSS fix. The number in use is less than or equal to the number in view.

Remarks

None

GNSSSatellites::numGLNInUse()

This function returns the number of GLONASS satellites used in computing navigation solution.

Syntax

#include “GNSS.h”

uint16_t num = GnssInfo.satellites.numGLNInUse(prnList);

Parameters

uint16_t*prnList

An address pointer points to an array of unsigned 16-bit values with number of array elements equal to **STGNSS_GLONASS_NCHAN** which is defined in “sti_gnss_lib.h”. If this pointer is not NULL then this array will contain the PRN list for GLONASS satellites used for GNSS fix by GNSS receiver. If a NULL pointer is given, then no PRN list will be returned.

Returns

A 16-bit unsigned integer to indicate the number of GLONASS satellites used by GNSS receiver in calculation for GNSS fix. The number in use is less than or equal to the number in view.

Remarks

None

GNSSSatellites::elevation()

This function returns the elevation angle for the specified satellite.

Syntax

#include “GNSS.h”

uint16_t angle = GnssInfo.satellites.elevation(constellation, prn);

Parameters

uint8_t constellation

The 1st argument is an 8-bit unsigned integer to specifyingthe satellite constellation. The available values are defined in “GNSS.h” and shown in below.

CONSTELLATION_GPS

CONSTELLATION_BD2

CONSTELLATION_GLONASS

uint16_t prn

The 2nd argument is A 16-bit unsigned integerdenoting PRN of the satellite. User can use numXXXInView() to get the valid PRN list before calling this function.

Returns

A 16-bit unsigned integer to indicate the integer part of elevation angle. The possible range is from 0 to 90.

Remarks

None

GNSSSatellites::azimuth()

This function returns the azimuth angle for the specified satellite.

Syntax

#include “GNSS.h”

uint16_t angle = GnssInfo.satellites.azimuth(constellation, prn);

Parameters

uint8_t constellation

The 1st argument is an 8-bit unsigned integer to specifying the satellite constellation. The available values are defined in “GNSS.h” and shown in below.

CONSTELLATION_GPS

CONSTELLATION_BD2

CONSTELLATION_GLONASS

uint16_t prn

The 2nd argument is A 16-bit unsigned integer denoting PRN of the satellite. User can use numXXXInView() to get the valid PRN before calling this function.

Returns

A 16-bit unsigned integer to indicate the integer part of azimuth angle. The possible range is from 0 to 359.

Remarks

None

GNSSSatellites::CNR()

This function returns the CNR for the specified satellite.

Syntax

#include “GNSS.h”

uint16_tsnr = GnssInfo.satellites.CNR(constellation, prn);

Parameters

uint8_t constellation

The 1st argument is an 8-bit unsigned integer to specifying the satellite constellation. The available values are defined in “GNSS.h” and shown in below.

CONSTELLATION_GPS

CONSTELLATION_BD2

CONSTELLATION_GLONASS

uint16_t prn

The 2nd argument is A 16-bit unsigned integer denoting PRN of the satellite. User can use numXXXInView() to get the valid PRN before calling this function.

Returns

A 16-bit unsigned integer to indicate the integer part of CNR. The normal range of CNR is from 0 to 55.

Remarks

None

3.3.12 GNSSTimeStamp

GNSSTimeStamp

Class related to external event trigger time stamping.

Public Member Functions

- GNSSTimeStamp(void)
- void setTrigCapture(bool enable, uint8_t trigMode, void (*callback))
- uint16_t numRecord(void)
- uint16_t idxRecord(void)
- bool push(TIME_STAMPING_STATUS_T ts)
- bool pop(void)
- void convertTimeStampToUTC(void)
- uint16_t year(void)
- uint8_t month(void)
- uint8_t day(void)
- uint8_t hour(void)
- uint8_t minute(void)
- uint8_t second(void)
- double fractional_sec(void)
- uint16_t formatUTCString(char* str)
- uint16_t formatGPSString(char* str)

Remarks

None

GNSSTimeStamp::setTrigCapture()

This function sets up Time-Stamp mechanism.

Syntax

#include “GNSS.h”

GnssInfo.timestamp.setTrigCapture(enable, trigMode, (void*)callback);

Parameters

bool enable

The 1st argument is ON/OFF setting for time-stamp trigger. The available values are defined in “GNSS.h” and shown in below.

TS_TRIG_ON

TS_TRIG_OFF

uint8_ttrigMode

The 2nd argument is an 8-bit unsigned integerspecifying rising-edge or falling-edgeof event trigger on pin GPIO10. The available values are defined in “GNSS.h” and shown in below.

TS_TRIG_RISING

TS_TRIG_FALLING

void (*callback)

The 3rd argument is an address pointer pointing to user-defined function which will be called when the event happens on pin GPIO10. There is one argument of type of structure **TIME_STAMPING_STATUS_T**(defined in “sti_gnss_lib.h”) in callback function.

Returns

None

Remarks

When user calls this function with 1stargument TRUE, this function will configure pin GPIO10 as input pin and disable the normal GPIO interrupt service routine for GPIO10, it also inserts the callback function to kernel space as part of interrupt service routine for time-stamp trigger. When user calls this function with 1st argument is FALSE, this function just remove the callback function from kernel.

GNSSTimeStamp::numRecord()

This function returns the number of triggered time-stamp records which are stored in internal FIFO of class object and waiting to be read.

Syntax

#include “GNSS.h”
uint16_t num = GnssInfo.timestamp.numRecord();

Parameters

None

Returns

A 16-bit unsigned integer to indicate the number of triggered time-stamp records in FIFO and waiting for read. Default depth of FIFO is 10.

Remarks

None

GNSSTimeStamp::idxRecord()

This function returns the index of current time-stamp record in the FIFO.

Syntax

#include “GNSS.h”
uint16_tidx = GnssInfo.timestamp.idxRecord();

Parameters

None

Returns

A 16-bit unsigned integer which equals to the index of current time-stamp record in FIFO.

Remarks

The GNSS receiver will generate an incremental trigger index for each triggered time-stamp record, serving as unique ID number associate with the time-stamp, up to 65535. This function gets the index number for the 1st unread record in FIFO.

GNSSTimeStamp::pop()

This function is used to get a new unread record from FIFO.

Syntax

#include “GNSS.h”

bool result = GnssInfo.timestamp.pop();

Parameters

None.

Returns

A boolean TRUE if the FIFO has at least one unread record, otherwise returns a boolean FALSE.

Remarks

After successfully calling pop(), the 1st unread time-stamp record in FIFO will be duplicated to an internal member with the same type of structure TIME_STAMPING_STATUS_T. Functions described later will get time-stamp information based on this internal member.

GNSSTimeStamp::convertTimeStampToUTC()

This function is converts time-stamp record to UTC format.

Syntax

#include “GNSS.h”
GnssInfo.timestamp.convertTimeStampToUTC();

Parameters

None.

Returns

None

Remarks

After calling convertTimeStampToUTC(), the information of time-stamp record kept in an internal member of type of structure TIME_STAMPING_STATUS_T will be converted to another internal member of type of structure UTC_TIME_T.

GNSSTimeStamp::year()

This function returns the year value of time-stamp record popped from FIFO.

Syntax

#include “GNSS.h”
uint16_t ts_year = GnssInfo.timestamp.year();

Parameters

None.

Returns

A 16-bit unsigned integer to indicate the year of the time-stamp record popped from FIFO.

Remarks

The function convertTimeStampToUTC() must be called prior to calling year() or user may get an incorrect year value.

GNSSTimeStamp::month()

This function returns the month value of time-stamp record popped from FIFO.

Syntax

#include “GNSS.h”

uint8_t ts_month = GnssInfo.timestamp.month();

Parameters

None.

Returns

An 8-bit unsigned integer to indicate the month of the time-stamp record popped from FIFO.

Remarks

The function convertTimeStampToUTC() must be called prior to calling month() or user may get an incorrect month value.

GNSSTimeStamp::day()

This function returns the day value of time-stamp record popped from FIFO.

Syntax

#include “GNSS.h”
uint8_t ts_day = GnssInfo.timestamp.day();

Parameters

None

Returns

An 8-bit unsigned integer to indicate the day of the time-stamp record popped from FIFO.

Remarks

The function convertTimeStampToUTC() must be called prior to calling day() or user may get an incorrect day value.

GNSSTimeStamp::hour()

This function returns the day value of time-stamp record popped from FIFO.

Syntax

#include “GNSS.h”

uint8_t ts_hour = GnssInfo.timestamp.hour();

Parameters

None

Returns

An 8-bit unsigned integer to indicate the hour of the time-stamp record popped from FIFO.

Remarks

The function convertTimeStampToUTC() must be called prior to calling hour() or user may get an incorrect hour value.

GNSSTimeStamp::minute()

This function returns the minute value of time-stamp record popped from FIFO.

Syntax

#include “GNSS.h”

uint8_t ts_minute = GnssInfo.timestamp.minute();

Parameters

None

Returns

An 8-bit unsigned integer to indicate the minute of the time-stamp record popped from FIFO.

Remarks

The function convertTimeStampToUTC() must be called prior to calling minute() or user may get an incorrect minute value.

GNSSTimeStamp::second()

This function returns the integer part of seconds of time-stamp record popped from FIFO.

Syntax

#include “GNSS.h”

uint8_t ts_second = GnssInfo.timestamp.second();

Parameters

None

Returns

An 8-bit unsigned integer to indicate the integer part of seconds of the time-stamp record popped from FIFO.

Remarks

The function convertTimeStampToUTC() must be called prior to calling second() or user may get an incorrect seconds.

GNSSTimeStamp::fractional_sec()

This function returns the fractional part of seconds of time-stamp record popped from FIFO.

Syntax

#include “GNSS.h”

double ts_frac_sec = GnssInfo.timestamp.fractional_sec();

Parameters

None

Returns

A double precision floating point value to indicate the fractional part of seconds of the time-stamp record popped from FIFO.

Remarks

The function convertTimeStampToUTC() must be called prior to calling fractional_sec() or user may get an incorrect value.

GNSSTimeStamp::formatUTCString()

This function converts the date/time of time-stamp record to a string in a pre-defined UTC format.

Syntax

#include “GNSS.h”
uint16_t sizeStr = GnssInfo.timestamp.formatUTCString(str);

Parameters

char* str
An address pointer points to a string buffer which will contain the string represents the date/time of time-stamp record in pre-defined UTC format.

Returns

A 16-bit unsigned integer to indicate the size of string.

Remarks

Precision of time-stamp is better than 100 nanoseconds. Here we defined an UTC format string with fractional part of seconds to sixth digit after the decimal point. Below is an example for the string: **2014-05-01 @ 10:42:33(+0.123456) PM**. User can change the format implemented in “GNSS.cpp” by their need.

GNSSTimeStamp::formatGPSString()

This function generates a string to represent the GPS date/time.

Syntax

#include “GNSS.h”
uint16_t sizeStr = GnssInfo.timestamp.formatGPSString(str);

Parameters

Char* str
An address pointer points to a string buffer which will contain the string represents the date/time of time-stamp record in GPS format.

Returns

A 16-bit unsigned integer to indicate the size of string.

Remarks

Precision of time-stamp provided by NavSpark is better than 100 nanoseconds. Here we defined an UTC format string with fractional part of seconds to ninth digit after the decimal point. Below is an example for the string: **123.4567 ms, week #36 since 1980**. User can change the format implemented in “GNSS.cpp” by their need.

3.3.13 Timer

TIMER

Class related to built-in H/W timers

Public Member Functions

- TIMER(void)
- TIMER(uint8_t tmrId)
- bool isIdle(void)
- void isr(void)
- uint8_t every(uint32_t period, void (*callback)(void))
- uint8_t every(uint32_t period, void (*callback)(void), uint16_t repeatCount)
- uint8_t after(uint32_t period, void (*callback)(void))
- bool expire(void)
- void stop(void)
- uint16_t remainTimes(void)

Remarks

None

TIMER::TIMER()

The constructor of class **TIMER**.

Syntax

#include “Timer.h”
TIMER Timer0(0);
TIMER Timer1(1);
TIMER Timer2(2);

Parameters

uint8_t tmrId
An 8-bit unsigned integer to specify the ID for timer, the legal values are 0, 1 and 2.

Returns

None

Remarks

Three pre-instantiated timer objects, Timer0/Timer1/Timer2 have been defined in “Timer.cpp” for built-in H/W timers. It is unnecessary for users to call the constructor manually.

TIMER::isIdle()

This function is used to check if timer is idle, not counting.

Syntax

```
#include “Timer.h”
booltmr0 = Timer0.isIdle();
bool tmr1 = Timer1.isIdle();
bool tmr2 = Timer2.isIdle();
```

Parameters

None

Returns

Boolean true if counting is completed, or boolean false in case of timer is counting.

Remarks

None

TIMER::every()

This function is used to start a repeatedly counting timer for repeated jobs.

Syntax

#include “Timer.h”
uint8_t tmr0 = Timer0.every(period, callback);
uint8_t tmr1 = Timer1.every(period, callback);
uint8_t tmr2 = Timer2.every(period, callback);

Parameters

uint32_t period

The 1st argument is the period in milliseconds that the timer will count down to zero and expire.

void (*callback)(void)

The 2nd argument is the callback function provided by user that NavSpark will execute when the timer expires.

Returns

An 8-bit unsigned integer.‘1’ means the configuration of timer was accepted, ‘0’ means timer is invalid.

Remarks

After calling this function the timer will start down-counting and expire after specified period, then jump to the callback function. After leaving callback function, the timer restarts counting and repeats such cycles unless calling stop() to terminate this operation. See example “demo_timer” for more detail.

TIMER::every()

This function is used to start a timer for set number of cycles.

Syntax

```
#include “Timer.h”
uint8_t tmr0 = Timer0.every(period, callback, repeatCount);
uint8_t tmr1 = Timer1.every(period, callback, repeatCount);
uint8_t tmr2 = Timer2.every(period, callback, repeatCount);
```

Parameters

uint32_t period

The 1st argument is the period in milliseconds that the timer will count down to zero and expire.

void (*callback)(void)

The 2nd argument is the callback function provided by user that NavSpark will execute when the timer expires.

uint16_t repeatCount

The 3rd argument is number of cycles that the timer should repeat.

Returns

An 8-bit unsigned integer.‘1’ means the configuration of timer was accepted, ‘0’ means timer is invalid.

Remarks

After calling this function the timer will start counting and expire after specified period, then jump to the callback function. After leaving callback function, the timer restarts counting and repeats such the cycle till the repeated cycles reach repeat count. See example “demo_timer” for more detail.

TIMER::after()

This function is used to start a timer for only one cycle count.

Syntax

```
#include “Timer.h”
uint8_t tmr0 = Timer0.after(period, callback);
uint8_t tmr1 = Timer1.after(period, callback);
uint8_t tmr2 = Timer2.after(period, callback);
```

Parameters

uint32_t period

The 1st argument is the period in milliseconds that the timer will count down to zero and expire.

void (*callback)(void)

The 2nd argument is the callback function provided by user that NavSpark will execute when the timer expires.

Returns

An 8-bit unsigned integer.‘1’ means the configuration of timer was accepted, ‘0’ means timer is invalid.

Remarks

This function is actually a polymorphism of every() with repeat Count is 1. See example “demo_timer” for more detail.

TIMER::expire()

This function is used to check if a timer already expired and is waiting for processing.

Syntax

```
#include “Timer.h”
bool tmr0 = Timer0.expire();
bool tmr1 = Timer1.expire();
bool tmr2 = Timer2.expire();
```

Parameters

None

Returns

Boolean true in case of timer expired, or boolean false for timer is idle or counting now.

Remarks

A dispatcher function, isrTimerFunc(), for timer interrupt service is defined in “Timer.cpp”. The Timer::expire() is used to identify which timer triggered interrupt, then user-defined callback function will be executed.

TIMER::stop()

This function is used to stop a timer from counting.

Syntax

```
#include “Timer.h”
Timer0.stop();
Timer1.stop();
Timer2.stop();
```

Parameters

None

Returns

None

Remarks

None

TIMER::remainTimes()

This function is used to query how many repeat cycles the timer need to run.

Syntax

#include “Timer.h”

uint16_t cycles = Timer0.remainTimes();

Parameters

None

Returns

A 16-bit unsigned integer to indicate the count of remaining cycles.

Remarks

For example, assuming one user enables timer 0 such that timer0 will expire every 300ms and repeat for 100 times, namely Timer0.every(300, callback, 100). After 1,500 ms, the user uses Timer0.remainTimes() to query how many cycles left now. The returned value is 95.

3.3.14 SPI_MasterSlave

SPI_MasterSlave

Class related to SPI master or slave interface.

Public Member Functions

- SPI_MasterSlave
- void config(uint8_t spiMode, uint32_t clkRate, bool en_cs1, bool en_cs2)
- void begin(void)
- void resetTx(void)
- void resetRx(void)
- size_t write(uint8_t data)
- size_t write(uint8_t *data, size_t size)
- void slaveSelect(uint8_t slv)
- size_t transfer(void)
- size_t transfer(size_t size)
- size_t available(void)
- int remaining(void)
- int read(void)
- uint8_t pick(size_t offset)
- void enableBufferForHostWrite(void)
- bool validBufferForHostRead(void)
- uint8_t copyDataToBufferForHostRead(void)
- void enableBufferForHostRead(void)
- void attachInterrupt(uint8_t type, void (*userFunc)(void))
- void detachInterrupt(uint8_t type)
- void isr (uint8_t type)

Remarks

None

SPI_MasterSlave::SPI_MasterSlave()

The constructor of class **SPI**.

Syntax

#include “SPI.h”
SPI_MasterSlave obj = SPI_MasterSlave (type);

Parameters

uint8_t type
An 8-bit unsigned integer selecting SPI master mode (type = 1) or SPI slave mode (type = 0). Calling this constructor without argument implies SPI master mode.

Returns

A SPI object.

Remarks

Two pre-instantiated SPI objects, **spiMater** and **spiSlave** have been defined in “SPI.cpp” by following code:

```
SPI_MasterSlave spiMaster      = SPI_MasterSlave (SPI_MASTER);
SPI_MasterSlave spiSlave       = SPI_MasterSlave (SPI_SLAVE);
```

SPI_MasterSlave::config()

This function is used to configure the SPI mode. And configure clock rate for SPI master only.

Syntax

#include “SPI.h”

spiMaster.config(spiMode, clkRate, en_cs1, en_cs2);

Parameters

uint8_t spiMode

An 8-bit unsigned integer to specify the SPI mode. Currently only mode 0 and 1 are supported.

uint32_t clkRate

A 32-bit unsigned integer to specify the rate of SPI clock in Hz.

bool en_cs1

A boolean value to specify if the SPI chip select 1 (GPIO22) is used to connect to remote device.

bool en_cs2

A boolean value to specify if the SPI chip select 2 (GPIO6) is used to connect to remote device.

Returns

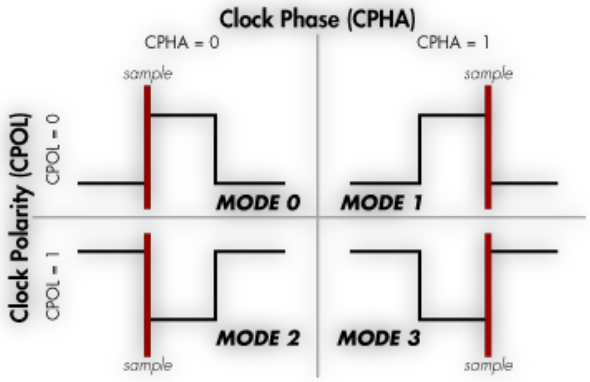
None

Remarks

There are four SPI modes shown in below and NavSpark supports mode 0 and 1.

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

SPI clock can operate at very high clock rate, limitation mostly depends on maximum SPI clock rate accepted by the slave device.



SPI_MasterSlave::begin()

This function is used to setup the GPIO pins for SPI operation and perform necessary initialization for SPI H/W receiver.

Syntax

```
#include “SPI.h”
spiMaster.begin();
spiSlave.begin();
```

Parameters

None

Returns

None

Remarks

Currently NavSpark supports three SPI masters with separate chip select pins and one SPI slave, see table in below for more detail.

GPIO pin	NavSpark as SPI master	NavSpark as SPI slave
6	chip select 2 (output)	
22	chip select 1 (output)	
28	chip select 0 (output)	chip select (input)
29	SCK (output)	SCK (input)
30	MOSI (output)	MOSI (input)
31	MISO (input)	MISO (output)

SPI_MasterSlave::resetTx()

This function is used to reset the internal transmit buffer.

Syntax

#include “SPI.h”
spiMaster.resetTx();
spiSlave.resetTx();

Parameters

None

Returns

None

Remarks

A transmit buffer is implemented in “SPI.cpp” to transmit data; this function will reset the pointer back to beginning of buffer.

SPI_MasterSlave::resetRx()

This function is used to reset the internal receive buffer.

Syntax

#include “SPI.h”
spiMaster.resetRx();
spiSlave.resetRx();

Parameters

None

Returns

None

Remarks

A receive buffer is implemented in “SPI.cpp”to receive data; this function will reset the pointer back to beginning of buffer.

SPI_MasterSlave::write()

This function is used to put one byte of data to the internal transmit buffer.

Syntax

#include “SPI.h”
size_t wrSize = spiMaster.write(data);
size_t wrSize = spiSlave.write(data);

Parameters

uint8_t data
The argument is the one byte data to be transmitted.

Returns

This function will return 1 if the buffer is valid and accept the data, or return 0 for the buffer is invalid.

Remarks

None

SPI_MasterSlave::write()

This function is used to put multiple bytes of data to the internal transmit buffer.

Syntax

#include “SPI.h”
size_t wrSize = spiMaster.write(data, size);
size_t wrSize = spiSlave.write(data, size);

Parameters

uint8_t *data
The 1st argument is an address pointer points to the buffer which contains the data to be transmitted.
size_t size
The 2nd argument is the number of bytes from the beginning of **data** by which NavSpark should copy to internal transmit buffer.

Returns

A number of bytes (type size_t) to indicate how many bytes of data have been put into internal transmit buffer.

Remarks

None

SPI_MasterSlave::slaveSelect()

This function is used to choose which slave device to communicate with.

Syntax

#include “SPI.h”
spiMaster.slaveSelect(slv);

Parameters

uint8_tslv
An 8-bit unsigned integer to specify which slave device should be active during later SPI communication. Valid values are 0, 1, 2 for currently supported3 chip select pins.

Returns

None

Remarks

This function is effective for SPI master and the default chip select is for slave 0.

SPI_MasterSlave::transfer()

This function is for master to transmit data to slave, and receive data from slave simultaneously.

Syntax

#include “SPI.h”
size_t num = spiMaster.transfer();
size_t num = spiMaster.transfer(size);

Parameters

The 1st argument is a type size_t value to specify the number of bytes to be sent from the position of buffer specified by transmit pointer, the transmit pointer can be reset to the beginning of buffer by SPI::resetTx() and incremented by the bytes transmitted over SPI bus. If no argument is given, the size of transmit data remaining in the buffer will be used internally.

Returns

A type size_t value to indicate how many bytes of data are transmitted/received during this transaction.

Remarks

Due to H/W limitation, a single transaction can only transfer maximum 8 bytes of data. For transfer of large data, user may need to call SPI::transfer() several times. A simple example is shown in below.

```
spiMaster.resetTx();
spiMaster.resetRx();
for (i = 0; i < 16; i++) { /* write 16 bytes of data to transmit buffer */
spiMaster.write(i);
}

io_cnt = spiMaster.transfer(16); /* inform SPI total 16 bytes to be transmitted */
while(io_cnt != 16) {
io_cnt += spiMaster.transfer();
}
```

Currently, maximum data transfer size at one transfer is 16 bytes.

SPI_MasterSlave::available()

This function is used to check if any incoming data stored in buffer.

Syntax

#include “SPI.h”
int num = spiMaster.avaiable();
int num = spiSlave.avaiable();

Parameters

None

Returns

A value of type size_t to indicate how many bytes of unread data are received in buffer.

Remarks

None

SPI_MasterSlave::remaining()

This function is used to check if any outgoing data not transmitted yet.

Syntax

#include “SPI.h”
int num = spiMaster.remaining();
int num = spiSlave.remaining();

Parameters

None

Returns

An integer value to indicate how many bytes of outgoing data are in transmit buffer but not transmitted yet.

Remarks

None

SPI_MasterSlave::read()

This function is used to pop received data from internal buffer.

Syntax

```
#include “SPI.h”
int rxd = spiMaster.read();
int rxd = spiSlave.read();
```

Parameters

None

Returns

An integer with its value equals to the 1st bytes of unread data in received buffer.

Remarks

It is recommended that user uses SPI_MasterSlave::available() to check if there is any unread received data in buffer followed by SPI_MasterSlave::read(). See a simple example shown in below.

```
while (spiMaster.available()) {
        rxd = spiMaster.read();
        // parsing rxd by your need here
}
```

SPI_MasterSlave::pick()

This function is used to pop received data from internal buffer.

Syntax

#include “SPI.h”
int rxd = spiMaster.pick(offset);
int rxd = spiSlave.pick(offset);

Parameters

size_t offset
The 1st argument is a value to specify the position of internal received buffer from which one byte of data will be extracted.

Returns

An integer with its value equals to the data on the specified position of buffer.

Remarks

None

SPI_MasterSlave::enableBufferForHostWrite()

This function is used for SPI slave interface. See section 4-1 for more detail.

Syntax

#include “SPI.h”
spiSlave.enableBufferForHostWrite();

Parameters

None

Returns

None

Remarks

Calling this function will set bit-1 of MAIN_REG, remote SPI host can read this register through SPI protocol to know that the WBUF is available to be written by SPI host.

SPI_MasterSlave::validBuffForHostRead()

This function is used for SPI slave interface. See section 4-1 for more detail.

Syntax

#include “SPI.h”
bool result = spiSlave.validBufferForHostRead();

Parameters

None

Returns

Boolean true if buffer empty all data read by SPI host, or boolean false if there is still some data not read by SPI host.

Remarks

Calling this function to make sure the buffer is empty ready for refill, then uses SPI_MasterSlave::write() to put data to the internal FIFO for SPI host, followed by calling SPI_MasterSlave::copyDataToBufferForHostRead() and SPI_MasterSlave::enableBufferForHostRead()to inform SPI host that new data is waiting to be read.

SPI_MasterSlave::copyDataToBufferForHostRead()

This function is used for SPI slave interface. See section 4-1 for more detail.

Syntax

#include “SPI.h”
uint8_t num = spiSlave.copyDataToBufferForHostRead();

Parameters

None

Returns

An 8-bit unsigned integer to indicate how many bytes of data have been moved from internal transmit FIFO to RBUF_REG.

Remarks

Calling this function will move the specified number of data from internal transmit FIFO to the RBUF_REG and set the number to RBUF_STS_REG. It also update the transmit pointer and size of transmit FIFO.

SPI_MasterSlave::enableBufferForHostRead()

This function is used for SPI slave interface. See section 4-1 for more detail.

Syntax

```
#include “SPI.h”
spiSlave.enableBufferForHostRead();
```

Parameters

None

Returns

None

Remarks

Calling this function will set the bit-2 of MAIN_REG. The SPI host can read MAIN_REG and check this bit to identify if it can issue a new read transaction.

SPI_MasterSlave::attachInterrupt()

This function is used for SPI slave interface. See section 4-1 for more detail.

Syntax

#include “SPI.h”

spiSlave.attachInterrupt(type, userFunc);

Parameters

uint8_t type

The 1st argument is an 8-bit unsigned integer to specify which ISR function is attached, the legal values are list in below.

IRQ_SPI_SLAVE_HOST_READ_DONE

IRQ_SPI_SLAVE_HOST_WRITE_DONE

void (*userFunc)(void)

The 2nd argument is the entry of user-provided ISR to be hooked.

Returns

None

Remarks

User can apply this function to hook their ISR to do extra task after SPI host read/write data from/to hardware FIFO. A simple example is given in below.

```
voidusrTask1(void)
{
    // add your code here for SPI host read done
}
voidusrTask2(void)
{
    // add your code here for SPI host write done
}

setup()
{
    spiSlave.attachInterrupt(IRQ_SPI_SLAVE_HOST_READ_DONE, usrTask1);
    spiSlave.attachInterrupt(IRQ_SPI_SLAVE_HOST_WRITE_DONE, usrTask2);
}
```

SPI_MasterSlave::detachInterrupt()

This function is used for SPI slave interface. See section 4-1 for more detail.

Syntax

#include “SPI.h”
spiSlave.detachInterrupt(type);

Parameters

uint8_t type

The 1st argument is an 8-bit unsigned integer to specify which ISR function to be de-attached, the legal values are list in below.

IRQ_SPI_SLAVE_HOST_READ_DONE
IRQ_SPI_SLAVE_HOST_WRITE_DONE

Returns

None

Remarks

None

SPI_MasterSlave::isr()

This function is used for SPI slave interface. See section 4-1 for more detail.

Syntax

#include “SPI.h”
spiSlave.isr(type);

Parameters

uint8_t type

The 1st argument is an 8-bit unsigned integer to specify which interrupt case is triggered now. The possible values are listed in below.

IRQ_SPI_SLAVE_RESET
IRQ_SPI_SLAVE_HOST_READ_DONE
IRQ_SPI_SLAVE_HOST_WRITE_DONE

Returns

None

Remarks

When any interrupt associated with SPI slave is trigged, ISR dispatcher “hwISRFunc()” defined in “wiring_intr.c”will be entered, and the dispatcher will call “isrSPISlaveFunc()” which is defined in “SPI.cpp” with proper argument.

3.3.15 SPIClass

SPIClass

Class related to SPI master interface. This class was created to have members that compatible with Arduino SPI library.

Public Member Functions

- SPIClass
- void begin(void)
- void end(void)
- uint8_t transfer(uint8_t data)
- uint16_t transfer16(uint16_t data)
- void beginTransaction(SPISettings settings)
- void endTransaction(void)

Remarks

None

SPISettings

Class to configure SPI master port for SPI device used in SPI.beginTransaction in SPIClass. Descriptions of SPIsettings are in SPI.beginTransaction.

SPIClass::SPIClass()

The constructor of class **SPIClass**.

Syntax

#include “SPI.h”

SPIClass obj = SPIClass(type);

Parameters

uint8_t type

An 8-bit unsigned integer selecting SPI master mode (type = 1) or SPI slave mode (type = 0). Calling this constructor without argument implies SPI master mode. Currently, only type 1 is supported in this class.

Returns

A SPI object.

Remarks

One pre-instantiated SPI objects, **SPI** has been defined in “SPI.cpp” by following code:

```
SPIClass  SPI      = SPIClass(SPI_MASTER);
```

SPIClass::begin()

This function is used to setup the GPIO pins for SPI operation and perform necessary initialization for SPI H/W receiver.

Syntax

#include “SPI.h”
SPI.begin();

Parameters

None

Returns

None

Remarks

Currently NavSpark supports one SPI master with chip select pins other than GPIO 28,29,30,31.

GPIO pin	NavSpark as SPI master	Notes
General I/O	chip select (output)	GPIO other than 28, 29,30,31. Use digitalWrite(GPIO, LOW) and digitalWrite(GPIO, HIGH) as select and de-select chip select
29	SCK (output)	
30	MOSI (output)	
31	MISO (input)	

SPIClass::end()

This function is used to disable the SPI bus.

Syntax

#include “SPI.h”
SPI.end();

Parameters

None

Returns

None

Remarks

None

SPIClass::transfer()

This function is for master to transmit data to slave, and receive data from slave simultaneously.

Syntax

#include “SPI.h”

uint8_t rxdata = SPI.transfer(data);

Parameters

uint8_t data

The argument is a type of uint8_t data value to be sent over the SPI bus.

Returns

The received data of type uint8_t from the SPI bus.

Remarks

None

SPIClass::transfer16()

This function is for master to transmit data to slave, and receive data from slave simultaneously.

Syntax

#include “SPI.h”

uint16_t rxdata = SPI.transfer16(data);

Parameters

Uint16_t data

The argument is a type of uint_16 data value to be sent over the SPI bus.

Returns

The received data of type uint16_t from the SPI bus.

Remarks

None

SPIClass::beginTransaction()

This function is for master to initialize the SPI bus using object of SPISettings class.

Syntax

#include “SPI.h”

SPI.beginTransaction(Settings);

Parameters

SPISettings Settings

The argument is a type of SPISettings to initialize the SPI bus.

Returns

None

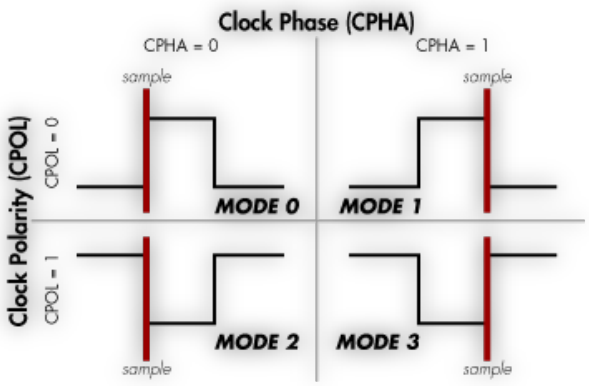
Remarks

SPISettings has 3 parameters, syntax: SPISettings Settting(SPI clock, dataOrder, dataMode), SPI clock, data order and data mode. SPI clock is the clock rate over the SPI bus, ex. 1Mhz, use 1000000. Data order can be MSBFIRST. Data mode can be SPI_MODE0 and SPI_MODE1.

There are four SPI modes shown in below and NavSpark supports mode 0 and 1.

Mode	CPOL	CPHA
0	0	0
1	0	1
2	1	0
3	1	1

SPI clock can operate at very high clock rate, limitation mostly depends on maximum SPI clock rate accepted by the slave device.



SPIClass::endTransaction()

This function is for master to stop using SPI bus.

Syntax

#include “SPI.h”
SPI.endTransaction();

Parameters

None

Returns

None

Remarks

None

3.3.16 TwoWire_MasterSlave

TwoWire_MasterSlave

Class related to the interface for two-wire master or slave.

Public Member Functions

- TwoWire_MasterSlave
- void config(uint32_t clkRate)
- void reset(void)
- void begin(uint8_t addr)
- void setTransmitDeviceAddr(uint8_t addr)
- void setReceiveDeviceAddr(uint8_t addr)
- uint16_t endTransmission(bool xmitStop)
- uint16_t readDevice(uint16_t quantity, bool xmitStop)
- uint16_t readDeviceFromOffset(uint8_t devOffset, uint16_t quantity, bool xmitStop)
- size_t write(uint8_t data)
- size_t write(uint8_t *data, size_t size)
- size_t writeAtOffset(uint8_t devOffset, uint8_t *data, size_t size)
- int available(void)
- int read(void)
- int peek(void)
- void onReceive(void (*function)(void))
- void onRequest(void (*function)(void))
- void isr(void)
- uint8_t requestFrom(uint8_t address, uint8_t quantity)
- uint8_t requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop)
- uint8_t beginTransmission (**uint8_t** address)

Remarks

None

TwoWire_MasterSlave::TwoWire_MasterSlave()

The constructor of class TwoWire_MasterSlave.

Syntax

#include “TwoWire.h”
TwoWire_MasterSlave obj = TwoWire_MasterSlave (type);

Parameters

uint8_t type
An 8-bit unsigned integer to specify the object is a 2-wire master (type = 1) or 2-wire slave (type = 0). Calling this constructor without argument implies a 2-wire master.

Returns

A 2-wire object.

Remarks

Two pre-instantiated 2-wire objects, **twMater** and **twSlave** have been defined in “TwoWire.cpp” by following code:

```
TwoWire_MasterSlave twMaster = TwoWire_MasterSlave (TWOWIRE_MASTER);
TwoWire_MasterSlave twSlave  = TwoWire_MasterSlave (TWOWIRE_SLAVE);
```

TwoWire_MasterSlave::config()

This function is used to configure clock rate on 2-wire bus.

Syntax

#include “TwoWire.h”
twMaster.config(clkRate);

Parameters

uint32_tclkRate
A 32-bit unsigned integer to specify the rate of clock of 2-wire.

Returns

None

Remarks

Up to 10MHz clock rate can be supported by the 2-wire interface, but usable clock rate depends on loading of the connected device and device’s 2-wirecharacteristics.

TwoWire_MasterSlave::begin()

This function is used to perform the physical H/W initialization of 2-wire controller.

Syntax

```
#include “TwoWire.h”
twMaster.begin();
twSlave.begin(addr);
```

Parameters

uint8_t addr

The 1st argument is a 32-bit unsigned integer to specify the device address for operation in slave mode. This argument can be omitted for 2-wire master.

Returns

None

Remarks

Device address can be set to any 7-bit value when configured as a two-wire slave device. A new device address could be given by calling this function before data transmission. The default device address of two-wire slave mode is 0x3C.

TwoWire_MasterSlave::setTransmitDeviceAddr()

This function is used to set the device address for sending data over 2-wire master mode.

Syntax

#include “TwoWire.h”
twMaster.setTransmitDeviceAddr(addr);

Parameters

uint8_t addr

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is usedby 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

Returns

None

Remarks

None

TwoWire_MasterSlave::setReceiveDeviceAddr()

This function is used to set the device address for receiving of data over 2-wire master mode.

Syntax

#include “TwoWire.h”
twMaster.setReceiveDeviceAddr(addr);

Parameters

uint8_t addr

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire mastet to select which device to perform read operation and the valid value is from 0 to 127.

Returns

None

Remarks

None

TwoWire_MasterSlave::endTransmission()

This function is used for 2-wire master to transmit data over 2-wire bus.

Syntax

#include “TwoWire.h”
twMaster.endTransmission(xmitStop);

Parameters

boolxmitStop

The 1st argument is a boolean value to specify if a STOP bit should be added after the last bit of data. If this argument is omitted, it implies a TRUE value is given and STOP bit will be added.

Returns

A 16-bit unsigned integer to indicate how many bytes of data were accepted by the remote device.

Remarks

This function will read all data in internal transmit FIFO which was filled by TwoWire_MasterSlave::write() and send them over the 2-wire bus, the return value indicate the exact number of bytes of data were accepted by remote device.

TwoWire_MasterSlave::readDevice()

This function is for 2-wire master to read data from remote device.

Syntax

#include “TwoWire.h”

uint16 num = twMaster.readDevice(quantity, xmitStop);

Parameters

uint16_t quantity

The 1st argument is a unsigned 16-bit value to specify the number of bytes to be read from device.

boolxmitStop

The 2ndargument is a boolean value to specify if a STOP bit should be added after the last bit of data. If this argument is omitted, it implies a TRUE value is given and a STOP bit will be added.

Returns

A 16-bit unsigned integer to indicate how many bytes of data read from the remote device.

Remarks

This function will perform direct read operation from remote device and save those data into internal receive FIFO. If the available space of FIFO is less than **quantity**, this function will only read data with size of available space of FIFO.

TwoWire_MasterSlave::readDeviceFromOffset()

This function is for 2-wire master to read data with specified offset from remote device.

Syntax

#include “TwoWire.h”

uint16 num = twMaster.readDeviceFromOffset(devOffset, quantity, xmitStop);

Parameters

uint8_t devOffset

The 1st argument is an 8-bit unsigned integer to specify the offset from which the data should be read from remote device.

uint16_t quantity

The 2ndargument is A 16-bit unsigned integer to specify the number of bytes to be read from device.

boolxmitStop

The 3rdargument is a boolean value to specify if a STOP bit should be added after the last bit of data.

Returns

A 16-bit unsigned integer to indicate how many bytes of data read from the remote device.

Remarks

This function is composed of two operations: the 1st operation is a write operation to notify remote device where to start the reading, the 2nd operation is the read operation just like “TwoWire_MasterSlave::readDevice()”.

TwoWire_MasterSlave::write()

This function puts1-byte data to internal transmit FIFO (for both slave and master).

Syntax

```
#include “TwoWire.h”
size_t num = twMaster.write(data);
size_t num = twSlave.write(data);
```

Parameters

uint8_tdata

The 1st argument is an 8-bit unsigned integer data to be transmitted.

Returns

In case of master mode, ‘1’ means the data was transmitted on bus successfully or ‘0’ means transmission failed. In case of slave mode, ‘1’ means the data was put to transmit FIFO successfully and ‘0’ means the FIFO rejected.

Remarks

None

TwoWire_MasterSlave::write()

This function is used to put data from source buffer to internal transmit FIFO(for both master and slave)

Syntax

```
#include “TwoWire.h”
size_t num = twMaster.write(data, size);
size_t num = twSlave.write(data, size);
```

Parameters

uint8_t *data

The 1st argument is a pointer points to an array of unsigned 8-bit data.

size_t size

The 2nd argument is a value to specify the number of data in bytes should be pushed to transmit FIFO.

Returns

In case of master mode, a value of type size_t to indicate the number of data transmitted successfully. In case of slave mode, this value only reflects the number of data to be put to transmit FIFO.

Remarks

None

TwoWire_MasterSlave::writeAtOffset()

This function is used to write data to remote slave at specified starting offset.

Syntax

#include “TwoWire.h”

size_t num = twMaster.writeAtOffset(devOffset, data, size);

Parameters

uint8_t devOffset

The 1st argument is an 8-bit unsigned integer to specify the offset of slave from which the data should be written.

uint8_t *data

The 1st argument is a pointer points to an array of unsigned 8-bit data.

size_t size

The 2nd argument is a value to specify the number of data in bytes to be written to remote slave.

Returns

A value of type size_t to indicate how many bytes of data have been written to remote slave.

Remarks

See “section 4-2” for more detail.

TwoWire_MasterSlave::available()

This function is used to query number of data received.

Syntax

```
#include “TwoWire.h”
int num = twMaster.available();
int num = twSlave.available();
```

Parameters

None

Returns

An integer value to indicate number of data in bytes received in receive FIFO.

Remarks

None

TwoWire_MasterSlave::read()

This function is used to get the data from received FIFO.

Syntax

#include “TwoWire.h”
intdata = twMaster.read();
intdata = twSlave.read();

Parameters

None

Returns

An integer value equals to the 1st unread byte of data in receive FIFO. If there is no unread data in FIFO, integer ‘-1’ will be returned.

Remarks

None

TwoWire_MasterSlave::peek()

This function is used to peek the data from received FIFO.

Syntax

#include “TwoWire.h”
int data = twMaster.peek();
int data = twSlave.peek();

Parameters

None

Returns

An integer value equals to the 2nd unread byte of data in receive FIFO. If there is no data or only 1-byte of data left in receive FIFO, integer ‘-1’ will be returned.

Remarks

Unlike “TwoWire_MasterSlave::read()”, this function does NOT change the value of internal read pointer, that is, user can get same data after calling “read()” twice.

TwoWire_MasterSlave::onReceive()

This function is used to hook the user-defined function which will be executed when data written by remote host is received.

Syntax

#include “TwoWire.h”
twSlave.onReceive(function);

Parameters

void (*function)(void)
The 1st argument is a pointer pointing to the entry of user-defined function.

Returns

None

Remarks

User may hook their function for post-processing after moving received data from RX_FIFO (see “section 4-2” for more detail) to internal buffer.

TwoWire_MasterSlave::onRequest()

This function is used to hook the user-defined function which will be executed when remote host request to transmit data.

Syntax

```
#include “TwoWire.h”
twSlave.onRequest(function);
```

Parameters

void (*function)(void)

The 1st argument is a pointer points to the entry of user-defined function.

Returns

None

Remarks

User may hook their function for pre-processing before moving data from internal buffer to TX_FIFO (see “section 4-2” for more detail).

TwoWire_MasterSlave::isr()

This function is used for 2-wire slave interface. See“section 4-2”for more detail.

Syntax

#include “TwoWire.h”
twSlave.isr();

Parameters

None

Returns

None

Remarks

When any interrupt associated with 2-wire slave is triggered, ISR dispatcher “hwISRFunc()” defined in “wiring_intr.c”will be entered, and the dispatcher will call “isrTwoWireSlaveFunc()”, which is defined in “TwoWire.cpp”.

TwoWire_MasterSlave::requestFrom()

This function is for 2-wire master to read data from remote device.

Syntax

#include “TwoWire.h”
twMaster.requestFrom(address, quanity);

Parameters

uint8_t address

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

uint8_t quanity

The 2nd argument is to specify the number of bytes to be read from device.

Returns

A 8-bit unsigned integer to indicate how many bytes of data read from the remote device.

Remarks

This function will perform direct read operation from remote device and save those data into internal receive FIFO. If the available space of FIFO is less than **quantity**, this function will only read data with size of available space of FIFO.

TwoWire_MasterSlave::requestFrom()

This function is for 2-wire master to read data from remote device.

Syntax

#include “TwoWire.h”

twMaster.requestFrom(address, quanity, sendStop);

Parameters

uint8_t address

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

uint8_t quanity

The 2nd argument is to specify the number of bytes to be read from device.

uint8_t sendStop

The 3rd argument is a boolean value to specify if a STOP bit should be added after the last bit of data. If this argument is omitted, it implies a TRUE value is given and a STOP bit will be added.

Returns

A 8-bit unsigned integer to indicate how many bytes of data read from the remote device.

Remarks

This function will perform direct read operation from remote device and save those data into internal receive FIFO. If the available space of FIFO is less than **quantity**, this function will only read data with size of available space of FIFO.

TwoWire_MasterSlave::requestFrom()

This function is for 2-wire master to read data from remote device.

Syntax

#include “TwoWire.h”

twMaster.requestFrom(address, quanity, sendStop);

Parameters

uint8_t address

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

uint8_t quanity

The 2nd argument is to specify the number of bytes to be read from device.

uint8_t sendStop

The 3rd argument is a boolean value to specify if a STOP bit should be added after the last bit of data. If this argument is omitted, it implies a TRUE value is given and a STOP bit will be added.

Returns

A 8-bit unsigned integer to indicate how many bytes of data read from the remote device.

Remarks

This function will perform direct read operation from remote device and save those data into internal receive FIFO. If the available space of FIFO is less than **quantity**, this function will only read data with size of available space of FIFO.

TwoWire_MasterSlave::beginTransmission()

This function is used to reset FIFO and set the I2C slave device with the given address as a 2-wire master.

Syntax

#include “TwoWire.h”

twMaster.beginTransmission(addr);

Parameters

uint8_t addr

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

Returns

None

Remarks

None

3.3.17 TwoWire

TwoWire

Class related to the interface for two-wire master or slave. This class was created to have members that compatible with Arduino wire filename and library.

Public Member Functions

- TwoWire
- void config(uint32_t clkRate)
- void begin(uint8_t addr)
- uint8_t requestFrom(uint8_t address, uint8_t quantity)
- uint8_t requestFrom(uint8_t address, uint8_t quantity, uint8_t sendStop)
- uint8_t beginTransmission (uint8_t address)
- uint16_t endTransmission()
- uint16_t endTransmission(bool xmitStop)
- size_t write(uint8_t data)
- size_t write(uint8_t *data, size_t size)
- int available(void)
- int read(void)
- void onReceive(void (*function)(void))
- void onRequest(void (*function)(void))

Remarks

The TwoWire class was created to be compatible with Arduino Wire library, Wire.h is the header and Wire is the object. 1.0.0 version of TwoWire has been moved to TwoWire_MasterSlave, shown in previous section. The TwoWire has the same members as TwoWire_MasterSlave to be backward compatible. Here in this section, we list Arduino compatible member functions and config.

TwoWire::TwoWire()

The constructor of class **TwoWire**.

Syntax

#include “Wire.h”

TwoWire obj = Wire();

Parameters

uint8_t type

An 8-bit unsigned integer to specify the object is a 2-wire master.

Returns

A 2-wire object.

Remarks

One pre-instantiated 2-wire object, **Wire** have been defined in “Wire.cpp” by following code:

TwoWire Wire = TwoWire();

TwoWire::config()

This function is used to configure clock rate on 2-wire bus.

Syntax

#include “Wire.h”

Wire.config(clkRate);

Parameters

uint32_tclkRate

A 32-bit unsigned integer to specify the rate of clock of 2-wire.

Returns

None

Remarks

Up to 10MHz clock rate can be supported by the 2-wire interface, but usable clock rate depends on loading of the connected device and device’s 2-wire characteristics.

TwoWire::begin()

This function is used to perform the physical H/W initialization of 2-wire controller.

Syntax

#include “Wire.h”
Wire.begin();
Wire.begin(addr);

Parameters

uint8_t addr

The 1st argument is a 32-bit unsigned integer to specify the device address for operation in slave mode. This argument can be omitted for 2-wire master.

Returns

None

Remarks

Device address can be set to any 7-bit value when configured as a two-wire slave device. A new device address could be given by calling this function before data transmission. The default device address of two-wire slave mode is 0x3C.

TwoWire:: requestFrom()

This function is for 2-wire master to read data from remote device.

Syntax

#include “Wire.h”

Wire.requestFrom(address, quanity);

Parameters

uint8_t address

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

uint8_t quanity

The 2ndargument is to specify the number of bytes to be read from device.

Returns

A 8-bit unsigned integer to indicate how many bytes of data read from the remote device.

Remarks

This function will perform direct read operation from remote device and save those data into internal receive FIFO.

If the available space of FIFO is less than **quantity**, this function will only read data with size of available space of FIFO.

TwoWire:: requestFrom()

This function is for 2-wire master to read data from remote device.

Syntax

#include “Wire.h”

Wire.requestFrom(address, quanity, sendStop);

Parameters

uint8_t address

The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

uint8_t quanity

The 2nd argument is to specify the number of bytes to be read from device.

uint8_t sendStop

The 3rd argument is a boolean value to specify if a STOP bit should be added after the last bit of data. If this argument is omitted, it implies a TRUE value is given and a STOP bit will be added.

Returns

A 8-bit unsigned integer to indicate how many bytes of data read from the remote device.

Remarks

This function will perform direct read operation from remote device and save those data into internal receive FIFO. If the available space of FIFO is less than **quantity**, this function will only read data with size of available space of FIFO.

TwoWire::beginTransmission()

This function is used to reset FIFO and set the I2C slave device with the given address as a 2-wire master.

Syntax

#include “Wire.h”
Wire.beginTransmission(addr);

Parameters

uint8_t addr
The 1st argument is an 8-bit unsigned integer to specify the 7-bit device address of remote device. This value is used by 2-wire master to select which device to perform write operation and the valid value is from 0 to 127.

Returns

None

Remarks

None

TwoWire::endTransmission()

This function is used for 2-wire master to transmit data over 2-wire bus..

Syntax

```
#include “Wire.h”
Wire.endTransmission(xmitStop);
```

Parameters

bool xmitStop

The 1st argument is a boolean value to specify if a STOP bit should be added after the last bit of data. If this argument is omitted, it implies a TRUE value is given and STOP bit will be added.

Returns

A 16-bit unsigned integer to indicate how many bytes of data were accepted by the remote device.

Remarks

This function will read all data in internal transmit FIFO which was filled by Wire::write() and send them over the 2-wire bus, the return value indicate the exact number of bytes of data were accepted by remote device.

TwoWire::write()

This function puts 1-byte data to internal transmit FIFO (for both slave and master). And start write operation on 2-wire bus (for 2-wire master only).

Syntax

#include “Wire.h”
size_t num = Wire.write(data);

Parameters

uint8_tdata
The 1st argument is an 8-bit unsigned integer data to be transmitted.

Returns

In case of master mode, ‘1’ means the data was transmitted on bus successfully or ‘0’ means transmission failed. In case of slave mode, ‘1’ means the data was put to transmit FIFO successfully and ‘0’ means the FIFO rejected.

Remarks

The behavior of this function differs for 2-wire master and 2-wire slave mode. When in 2-wire master mode, calling this function will put data into transmit FIFO followed by immediate data transmission on 2-wire bus, however, there is no data transmission on bus when in 2-wire slave mode.

TwoWire::write()

This function is used to put data from source buffer to internal transmit FIFO (for both master and slave). Start write operation on 2-wire bus (for 2-wire master only).

Syntax

#include “Wire.h”

size_t num = Wire.write(data, size);

Parameters

uint8_t *data

The 1st argument is a pointer points to an array of unsigned 8-bit data.

size_t size

The 2nd argument is a value to specify the number of data in bytes should be pushed to transmit FIFO.

Returns

In case of master mode, a value of type size_t to indicate the number of data transmitted successfully. In case of slave mode, this value only reflects the number of data to be put to transmit FIFO.

Remarks

The behavior of this function differs for 2-wire master and 2-wire slave mode. In master mode, calling this function will put data into transmit FIFO followed by immediate data transmission on 2-wire bus. For slave mode, there is no data transmission.

TwoWire::available()

This function is used to query number of data received.

Syntax

#include “Wire.h”
int num = Wire.available();

Parameters

None

Returns

An integer value to indicate number of data in bytes received in receive FIFO.

Remarks

None

TwoWire::read()

This function is used to get the data from received FIFO.

Syntax

#include “Wire.h”

Int data = Wire.read();

Parameters

None

Returns

An integer value equals to the 1st unread byte of data in receive FIFO. If there is no unread data in FIFO, integer ‘-1’ will be returned.

Remarks

None

TwoWire::onReceive()

This function is used to hook the user-defined function which will be executed when data written by remote host is received.

Syntax

#include “Wire.h”
Wire.onReceive(function);

Parameters

void (*function)(void)
The 1st argument is a pointer pointing to the entry of user-defined function.

Returns

None

Remarks

User may hook their function for post-processing after moving received data from RX_FIFO (see “section 4-2” for more detail) to internal buffer.

TwoWire::onRequest()

This function is used to hook the user-defined function which will be executed when remote host request to transmit data.

Syntax

#include “Wire.h”
Wire.onRequest(function);

Parameters

void (*function)(void)
The 1st argument is a pointer points to the entry of user-defined function.

Returns

None

Remarks

User may hook their function for pre-processing before moving data from internal buffer to TX_FIFO (see “section 4-2” for more detail).

TwoWire::isr()

This function is used for 2-wire slave interface. See“section 4-2”for more detail.

Syntax

#include “Wire.h”
Wire.isr();

Parameters

None

Returns

None

Remarks

When any interrupt associated with 2-wire slave is triggered, ISR dispatcher “hwISRFunc()” defined in “wiring_intr.c”will be entered, and the dispatcher will call “isrWireSlaveFunc ()”, which is defined in “Wire.cpp”.

3.3.18 HardwareSerial

HardwareSerial

Class related to UART interface.

Public Member Functions

- HardwareSerial
- void config(uint8_t word_length, uint8_t stop_bit, uint8_t party_check)
- void begin(uint32_t baudrate)
- void end(void)
- int available(void)
- int read(void)
- int peek(void)
- void flush(void)
- size_t write(uint8_t value)
- size_t write(uint8_t *buffer, size_t size)
- size_t print(const char str[])
- void isrRx(void)
- void taskTx(void)

Remarks

None

HardwareSerial::HardwareSerial()

The constructor of class **HardwareSerial**.

Syntax

#include “HardwareSerial.h”
HardwareSerial object= HardwareSerial(port);

Parameters

uint8_t port
The 1st argument is an 8-bit unsigned integer to specify which UART port is used.

Returns

None

Remarks

Two hardware UARTs are supported. See below table for pin out for UART 1 and 2.

Port	Direction	Pin Out	GNSS Mode	MCU Mode
UART 1	RX	N/A	Reserved for NMEA	Serial1
	TX	TXD0		
UART 2	RX	GPIO1	Serial	Serial
	TX	GPIO2		

Two pre-instantiated UART objects, **Serial** and **Serial1,** have been defined in “HardwareSerial.cpp” by following code:

```
HardwareSerial Serial = HardwareSerial(CONSOLE_PORT_ID);
HardwareSerial Serial1 = HardwareSerial(NMEA_PORT_ID);
```

When binary image links with GNSS library, UART 1will be used to output NMEA messages and communication of binary commands, user SHOULD avoid using UART1 in this case.

When binary image links MCU mode, the UART 1 can be released to use as Serial1 through the micro USB connector. On the pins, only UART 1 TX is accessible. Due to 3.3V LVTTL nature of the UART, an external UART to RS232 level shifter may be needed to connect to traditional COM port of computer.

HardwareSerial::config()

This function is used to configure UART parameters.

Syntax

#include “HardwareSerial.h”
Serial.config(word_length, stop_bit, party_check);
Serial1.config(word_length, stop_bit, party_check);

Parameters

uint8_t word_length

The 1st argument is an 8-bit unsigned integer to specify number of bitsin one word, the possible values are listed below:

STGNSS_UART_5BITS_WORD_LENGTH
STGNSS_UART_6BITS_WORD_LENGTH
STGNSS_UART_7BITS_WORD_LENGTH
STGNSS_UART_8BITS_WORD_LENGTH

uint8_t stop_bit

The 2nd argument is an 8-bit unsigned integer to specify number of stop bits, the possible values are listed in below:

STGNSS_UART_1STOP_BITS
STGNSS_UART_2STOP_BITS

uint8_t party_check

The 3rd argument is an 8-bit unsigned integer to specify if party check is needed; the possible values are listed in below:

STGNSS_UART_NOPARITY
STGNSS_UART_ODDPARITY
STGNSS_UART_EVENPARITY

Returns

None

Remarks

This function won’t perform physical configuration to hardware, instead, it just set the values to internal variables. If this function is not called, the default parameters for UART is 8-N-1 (8-bit word length, no party check, 1 stop bit).

HardwareSerial::begin()

This function is used to configure UART baud rate and perform H/W initialization.

Syntax

#include “HardwareSerial.h”
Serial.begin(uint32_t baudrate);
Serial1.begin(uint32_t baudrate);

Parameters

uint32_tbaudrate
The 1st argument is a 32-bit unsigned integer to specify the baud rate;valid range is from 4,800 to 460,800. If this argument is omitted, the default baud rate is determined by compiler configuration “build.baudrate” defined in “boards.txt”.

Returns

None

Remarks

None

HardwareSerial::end()

This function is used to disable the object without destroying it, so user may reactivate it later by calling begin() again.

Syntax

#include “HardwareSerial.h”
Serial.end();
Serial1.end();

Parameters

None

Returns

None

Remarks

None

HardwareSerial::available()

This function is used to get the number of unread data received in receive buffer.

Syntax

```
#include “HardwareSerial.h”
int num = Serial.available();
int num = Serial1.available();
```

Parameters

None

Returns

A value of type size_t to indicate number of unread data in receive buffer.

Remarks

None

HardwareSerial::read()

This function is used to get the 1st unread data from receive buffer.

Syntax

```
#include “HardwareSerial.h”
intdata = Serial.read();
intdata = Serial1.read();
```

Parameters

None

Returns

An integer value equals to 1st byte of unread data from internal receive buffer. If no unread data exists or object is not enabled, -1 will be return.

Remarks

None

HardwareSerial::peek()

This function is used to peek the 2nd unread data from receive buffer without change anything.

Syntax

```
#include “HardwareSerial.h”
int data = Serial.peek();
int data = Serial1.peek();
```

Parameters

None

Returns

An integer value equals to the 2nd byte of unread data from internal receive buffer. If number of unread datain byte is less than 2 or object is not enabled, -1 will be return.

Remarks

None

HardwareSerial::flush()

This function is used to move all data in transmit buffer to H/W transmit FIFO immediately.

Syntax

#include “HardwareSerial.h”
Serial.flush();
Serial1.flush();

Parameters

None

Returns

None

Remarks

None

HardwareSerial::write()

This function is used to put one byte data to the internal transmit buffer.

Syntax

#include “HardwareSerial.h”
size_t num = Serial.write(value);
size_t num = Serial1.write(valie);

Parameters

uint8_t value
The 1st argument is an 8-bit unsigned integer for one byte data to be transmitted.

Returns

‘1’ in case of the **value** was copied to transmit buffer or otherwise ‘0’.

Remarks

This function will triggeran interrupt for UART hardware TX FIFO empty, that is, an interrupt will be generated when the hardware TX FIFO is empty and the “HardwareSerial::taskTx()” will be executed to move data from internal buffer to hardware TX FIFO.

HardwareSerial::write()

This function is used to put one byte data to the internal transmit FIFO.

Syntax

#include “HardwareSerial.h”
size_t num = Serial.write(data, size);
size_t num = Serial1.write(data, size);

Parameters

uint8_t *data

The 1st argument is a pointer points to an array of unsigned 8-bit integers which contains the data to be transmitted.

size_t size

The 2nd argument is number of data in bytes which this function will copy from the data array to internal transmit FIFO.

Returns

A value of type size_t to indicate the number of data in bytes has been written to the internal transmitFIFO.

Remarks

This function will trigger an interrupt for UART hardware TX FIFO empty, that is, an interrupt will be generated when the hardware TX FIFO is empty and the “HardwareSerial::taskTx()” will be executed to move data from I nternal buffer to hardware TX FIFO.

HardwareSerial::print()

This function is used to copy the contents of a string to the internal transmit FIFO.

Syntax

#include “HardwareSerial.h”
size_t num = Serial.print(str);
size_t num = Serial1.print(str);

Parameters

const char str[]
The 1st argument is a pointer to a string-type buffer. A string-type means there is 1-byte NULL terminator, 0x0 or ‘\0’, appended to the end of characters.

Returns

A value of type size_t to indicate the size of **str** excluding the 1-byte NULL terminator.

Remarks

This function will trigger an interrupt for UART hardware TX FIFO empty, that is, an interrupt will be generated when the hardware TX FIFO is empty and the “HardwareSerial::taskTx()” will be executed to move data from internal buffer to hardware TX FIFO.

HardwareSerial::isrRx()

This function is used to move data in H/W receive FIFO to class' internal receive buffer.

Syntax

#include "HardwareSerial.h"
Serial.isrRx();
Serial1.isrRx();

Parameters

None

Returns

None

Remarks

Normally the UART receiving circuitry will receive incoming data from remote host and put them into hardware RX FIFO waiting for CPU to read out, an interrupt will be generated automatically if RX FIFO is not empty for over a specified period. In this case the dispatcher, hwISRFunc(), which is defined in "wiring_intr.c"will be executed, followed by executing "isrSerialFunc()", and the "HardwareSerial::isrRx()" will be executed.

HardwareSerial::taskTx()

This function is used to move data in internal transmit buffer to H/W transmit FIFO.

Syntax

#include “HardwareSerial.h”
Serial.taskTx();
Serial1.taskTx();

Parameters

None

Returns

None

Remarks

Normally this function will be called by “taskSerialFunc()” insidethe interrupt service routine “hwISRFunc()”when the UART hardware TX FIFO is empty. This function may be called many times until there is no more data in internal transmit buffer, in this case this function will turn off the interrupt mechanism for the UART hardware TX FIFO empty.

3.3.19 Analog

Analog

Group for PWM and ADC.

Functions

- void analogPWMPeriod(uint8_t pin, uint32_t value)
- void analogPWMFreq(uint8_t pin, uint32_t value)
- analogWrite(uint8_t pin, uint16_t value)
- void analogADCClock(uint8_t pin, uint32_t value)*1
- uint16_t analogRead(uint8_t pin)*1

Remarks

*1: Commercial GPS/GNSS chipset typically has 2 ~ 5 bit ADC to sampling IF signal for baseband processing; higher bit ADC is non-essential for GNSS chipset.Yet an experimental 10bit 200Ksps ADC test block is embedded into Venus822A. The ADC is not intended as standard feature of Venus822A nor NavSpark. No testing of ADC is done at chip level, testing capable of detecting 0V and 3V is done only at NavSpark board level production. No ENOB, SFDR, SINAD, INL, DNL characteristics are available for the ADC. The ADC related functions are included here only for those who wish to use it for experiment at his own risk, and will not hold seller of the product nor SkyTraq responsible for ADC function not working to expectation or don't work at all.

analogPWMPeriod()

The function is used to configure PWM output period.

Syntax

#include “wiring_analog.h”
analogPWMPeriod(pin, value);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer to specify which GPIO pin should be configured as a PWM output, currently only 1 PWM supported for NavSpark, and this argument must be 20.

uint32_t value

The 2nd argument is a 32-bit unsigned integer to specify period of one cycle in nanoseconds during which PWM generates ‘H’ and ‘L’ levels. The allowed range for this value is from 1,000 to 500,000,000; equivalently 1MHz ~ 2Hz.

Returns

None

Remarks

The PWM won’t be configured physically until calling “analogPWMWrite()”.

analogPWMFreq()

The function is used to configure PWM output frequency.

Syntax

#include “wiring_analog.h”
analogPWMFreq(pin, value);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer to specify which GPIO pin should be configured as a PWM output, currently NavSpark supports only 1 PWM and this argument must be 20.

uint32_t value

The 2nd argument is a 32-bit unsigned integer to specify frequency in Hz of 50% duty cycle pulse. The allowed range for this value is from 2 to 1,000,000.

Returns

None

Remarks

The PWM won’t be configured physically until calling “analogPWMWrite()”.

analogWrite()

The function is used to configure the desired duty cycle of PWM output.

Syntax

#include “wiring_analog.h”
analogWrite(pin, value);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer to specify which GPIO pin should be configured as a PWM output, currently NavSpark supports only 1 PWM and this argument must be 20.

uint16_t value

The 2nd argument is a 16-bit unsigned integer to specify the duty cycle. 0 is for 0% duty cycle and 255 is for 100%.

Returns

None

Remarks

After calling this function, the configuration of H/W is done immediately and the PWM waveform appears on GPIO pin 20.

analogADCClock()*1

The function is used to configure the sampling frequency for ADC.

Syntax

#include “wiring_analog.h”
analogADCClock(pin, value);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer to specify which ADC pin to be configured. Currently only one ADC pin on NavSpark, and this argument must be 0.

uint32_t value

The 2nd argument is a 32-bit unsigned integer to specify the sampling clock rate used for ADC. The suggested range is from 200kHz to 10MHz.

Returns

None

Remarks

**1: Commercial GPS/GNSS chipset typically has 2 ~ 5 bit ADC to sampling IF signal for baseband processing; higher bit ADC is non-essential for GNSS chipset.Yet an experimental 10bit 200Ksps ADC test block is embedded into Venus822A. The ADC is not intended as standard feature of Venus822A nor NavSpark. No testing of ADC is done at chip level, testing capable of detecting 0V and 3V is done only at NavSpark board level production. No ENOB, SFDR, SINAD, INL, DNL characteristics are available for the ADC. The ADC related functions are included here only for those who wish to use it for experiment at his own risk, and will not hold seller of the product nor SkyTraq responsible for ADC function not working to expectation or don’t work at all.*

analogRead()*1

The function is used to read ADC measurement value.

Syntax

#include “wiring_analog.h”
uint16_t value = analogRead(pin);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer to specify which ADC pin to be read. Only one ADC pin on NavSpark,and this must be 0.

Returns

A 16-bit unsigned integer to indicate measurement result of ADC. The range of value is from 0 to 1023.

Remarks

ADC is 10-bit. When this function is called, one measurement is sampled and value returned. It takes about 20 CPU clock cycles for a single measurement.

**1: Commercial GPS/GNSS chipset typically has 2 ~ 5 bit ADC to sampling IF signal for baseband processing; higher bit ADC is non-essential for GNSS chipset.Yet an experimental 10bit 200Ksps ADC test block is embedded into Venus822A. The ADC is not intended as standard feature of Venus822A nor NavSpark. No testing of ADC is done at chip level, testing capable of detecting 0V and 3V is done only at NavSpark board level production. No ENOB, SFDR, SINAD, INL, DNL characteristics are available for the ADC. The ADC related functions are included here only for those who wish to use it for experiment at his own risk, and will not hold seller of the product nor SkyTraq responsible for ADC function not working to expectation or don’t work at all.*

3.3.20 Digital

Digital

Group for digital I/O functions.

Functions

- unsigned long pulseIn(uint8_t pin, uint8_t state, unsigned long timeout)
- void pinMode(uint8_t pin, uint8_t mode)
- void digitalWrite(uint8_t pin, uint8_t val)
- int digitalRead(uint8_t pin)
- void attachInterrupt(uint8_t pin, void (*taskFunc)(void), int mode)
- void detachInterrupt(uint8_t pin)
- void interrupts(void)
- void noInterrupts(void)
- void hwISRFunc(void)

Remarks

None

pulseIn()

The function is used to measure duration of a pulse.

Syntax

#include “wiring_pulse.h”

unsigned long time = pulseIn(pin, state, timeout);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer to specify which GPIO pin is used to measure the pulse.

uint8_tstate

The 2nd argument is an 8-bit unsigned integer to specify which level (HIGH or LOW) to be measured.

unsigned long timeout

The 3rd argument is an unsigned long integer to specify the timeout value in milliseconds.

Returns

An unsigned long integer to indicate the time period for the pulse detected with unit in millisecond. Its value depends on following situations:

- In case no pulse matching expected **state**is found, this value equals to 0.
- In case a pulse matching expected state is detected but it lasted longer than**timeout**period, this value equals **timeout**.
- In case a pulse matching expected state is detected, shorter than **timeout** period,then this value represent duration of the pulse.

Remarks

None

pinMode()

The function is used to configure I/O direction of GPIO pin.

Syntax

#include “wiring_digital.h”

pinMode(pin, mode);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer selecting which GPIO pin to be configured. The possible values are defined in “pins_arduino.h”.

uint8_tmode

The 2nd argument is an 8-bit unsigned integer configuring mode of the selected GPIO pin. There are three possible values:

INPUT

OUTPUT

SPECIAL

INPUT_PULLUP (Not supported by NavSpark)

Since many GPIO pins are multiplexed with other function such as SPI or 2-wire, “SPECIAL” is defined as a state for selecting those special functions.

Returns

None

Remarks

None

digitalWrite()

The function is used to set output level of GPIO pin.

Syntax

#include “wiring_digital.h”
digitalWrite(pin, val);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer selecting which GPIO pin to be set. The possible values are defined in “pins_arduino.h”.

uint8_tval

The 2nd argument is an 8-bit unsigned integer specifying the output level (HIGH or LOW) of the selected GPIO pin.

Returns

None

Remarks

If user did not configure the GPIO pin in OUTPUT mode, this function will return directly without any effect.

digitalRead()

The function is used to get the logic level on GPIO pin.

Syntax

#include “wiring_digital.h”
int val = digitalRead(pin);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer selecting which GPIO pin to be read. The possible values are defined in “pins_arduino.h”.

Returns

An integer representing logic level on the GPIO pin. There are three cases:

- HIGH：voltage greater than 2.7 volt is present on the selected pin
- LOW：voltage less than 0.3 volts is present on the selected pin
- -1：the selected pin is not in INPUT mode

Remarks

None

attachInterrupt()

The function is used to set an ISR for specified GPIO pin.

Syntax

#include “wiring_intr.h”
attachInterrupt(pin, taskFunc, mode);

Parameters

uint8_t pin

The 1st argument is an 8-bit unsigned integer to select which GPIO pin to be hooked with. The possible values are defined in “pins_arduino.h”.

void (*taskFunc)(void)

The 2nd argument is entry point to the user-provided function which will be called when the specified trigger happens on the GPIO pin.

int mode

The 3rd argument is an integer to specify how the ISR will be trigged. For now **RISING** and **FALLING** modes are supported.

Returns

None

Remarks

This function will return directly without any effect if the selected GPIO pin is not in INPUT mode.

detachInterrupt()

The function is used to remove an ISR for selected GPIO pin.

Syntax

#include “wiring_intr.h”
detachInterrupt(pin);

Parameters

uint8_t pin
The 1st argument is an 8-bit unsigned integer to select which GPIO pin to be removed. The possible values are defined in “pins_arduino.h”.

Returns

None

Remarks

None

interrupts()

The function is used to enable all ISRs for peripherals.

Syntax

#include “wiring_intr.h”
interrupts(void);

Parameters

None

Returns

None

Remarks

This function will enable the interrupt for peripherals, for now there are four peripherals can be turned on by this function: GNSS, TIMER, UART and GPIO.

noInterrupts()

The function is used to disable all ISRs for peripherals.

Syntax

#include “wiring_intr.h”
noInterrupts(void);

Parameters

None

Returns

None

Remarks

This function will disable the interrupt for peripherals, for now there are four peripherals can be turned off by this function: GNSS, TIMER, UART and GPIO.

3.3.21 SDClass

SDClass

The SD library allows for reading from and writing to SD card.

Public Member Functions

- SDClass
- boolean begin()
- boolean exist(const char *filepath)
- boolean mkdir(const char *filepath)
- File open(const char *filepath, uint_8 mode)
- boolean remove(const char *filepath)
- boolean rmdir(const char *filepath)

Remarks

None

SDClass::SDClass()

The constructor of class SDClass.

Syntax

#include “SD.h”
SDClass obj;

Parameters

None

Returns

None.

Remarks

A pre-instantiated SDClass objects, **SD** have been defined in “SD.cpp” by following code:

SDClass SD;

SDClass::begin()

Initialize the SD library and card. This begins of the SPI bus (digital pins 29, 30, 31 on NavSpark boards) and the chip select pin (pin 28 on NavSpark boards).

Syntax

```
#include “SD.h”
SD.begin();
```

Parameters

None

Returns

Boolean true on success or boolean false on failure.

Remarks

None

SDClass::exist()

Tests whether a file or directory exists on the SD card.

Syntax

#include “SD.h”

bool fileExist = SD.exist(filename);

Parameters

const char *filename

The name of the file to test for existence, which can include directories(delimited by forward-slashes, /)

Returns

Boolean true if the file or directory exists, bool false if not.

Remarks

None

SDClass::mkdir()

Create a directory on the SD card.

Syntax

#include “SD.h”

SD.mkdir (filename);

Parameters

const char *filename

The name of the directory to create (delimited by forward-slashes, /)

Returns

Boolean true if the creation of the directory succeeded, else boolean false if not.

Remarks

None

SDClass::open()

Opens the file whose name is specified. If the file is opened for writing, it will be created if it doesn't already exist (but the directory containing it must already exist).

Syntax

```
#include “SD.h”
File logFile = SD.open(filename,  mode);
```

Parameters

const char *filename

The name the file to open, which can include directories (delimited by forward slashes, /)

uint8_t mode (optional)

The mode in which to open the file, defaults to FILE_READ - byte. one of:
FILE_READ: open the file for reading, starting at the beginning of the file.
FILE_WRITE: open the file for reading and writing, starting at the end of the file.

Returns

A File object referring to the opened file; if the file couldn't be opened, this object will evaluate to false in a boolean context, i.e. you can test the return value with "if (f)".

Remarks

None

SDClass::remove()

Remove a file from the SD card.

Syntax

#include “SD.h”
SD.remove(filename);

Parameters

const char *filename
The name of the file to remove, which can include directories (delimited by forward-slashes, /)

Returns

Boolean true if the removal of the file succeeded, boolean false if not. (if the file didn't exist, the return value is unspecified).

Remarks

None

SDClass::rmdir()

Remove a directory from the SD card. The directory must be empty.

Syntax

#include “SD.h”

SD.rmdir(foldername);

Parameters

const char *foldername

The name of the directory to remove, with sub-directories separated by forward-slashes, /

Returns

Boolean true if the removal of the directory succeeded, boolean false if not. (if the directory didn't exist, the return value is unspecified).

Remarks

None

3.3.22 File

File

The File class allows for reading from and writing to individual files on the SD card.

Public Member Functions

- int available()
- void close()
- void flush()
- int peek()
- uint32_t position()
- size_t print(const char* str)
- size println(const char* str)
- boolean seek(uint32_t pos)
- uint32size()
- int read()
- size_t write(uint8_t)

Remarks

None

File::available()

Check if there are any bytes available for reading from the file.

Syntax

#include “SD.h”

file.available()

Parameters

File file

An instance of the File class (returned by SD.open())

Returns

The number of bytes available (int).

Remarks

None

File::close()

Close the file, and ensure that any data written to it is physically saved to the SD card.

Syntax

#include “SD.h”
file.close();

Parameters

File file
An instance of the File class (returned by SD.open())

Returns

None

Remarks

None

File::flush()

Ensures that any bytes written to the file are physically saved to the SD card. This is done automatically when the file is closed.

Syntax

```
#include “SD.h”
file.frush();
```

Parameters

File file

An instance of the File class (returned by SD.open())

Returns

None

Remarks

None

File::peek()

Read a byte from the file without advancing to the next one. That is, successive calls to peek() will return the same value, as will the next call to read()..

Syntax

#include “SD.h”
file.peek();

Parameters

File file
An instance of the File class (returned by SD.open())

Returns

The next byte (or character), or -1 if none is available.

Remarks

None

File::position()

Get the current position within the file (i.e. the location to which the next byte will be read from or written to).

Syntax

#include “SD.h”

file.position();

Parameters

File file

An instance of the File class (returned by SD.open())

Returns

The position within the file (unsigned long)the position within the file (unsigned long)

Remarks

None

File::print()

Print data to the file, which must have been opened for writing. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3').

Syntax

#include “SD.h”
file.print(data);
file.print(data, BASE);

Parameters

File file
An instance of the File class (returned by SD.open())

type data
The data to print (type can be char, byte, int, long, or string)

int BASE (optional)
The base in which to print numbers: BIN for binary (base 2), DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

print() will return the number of bytes written, though reading that number is optional

Remarks

None

File::println()

Print data, followed by a carriage return and newline, to the File, which must have been opened for writing. Prints numbers as a sequence of digits, each an ASCII character (e.g. the number 123 is sent as the three characters '1', '2', '3')..

Syntax

#include “SD.h”
file.println();
file.println(data);
file.print(data, BASE);

Parameters

File file
An instance of the File class (returned by SD.open())

type data
The data to print (type can be char, byte, int, long, or string)

int BASE (optional)
The base in which to print numbers: BIN for binary (base 2), DEC for decimal (base 10), OCT for octal (base 8), HEX for hexadecimal (base 16).

Returns

println() will return the number of bytes written, though reading that number is optional.

Remarks

None

File::seek()

Seek to a new position in the file, which must be between 0 and the size of the file (inclusive)..

Syntax

#include “SD.h”
file.seek(pos);

Parameters

File file
An instance of the File class (returned by SD.open())

uint32_t pos
The position to which to seek (unsigned long)

Returns

Boolean true for success, boolean false for failure.

Remarks

None

File::size()

Get the size of the file.

Syntax

#include “SD.h”

file.size();

Parameters

File file

An instance of the File class (returned by SD.open())

Returns

The size of the file in bytes (unsigned long)

Remarks

None

File::read()

Read a byte from the file.

Syntax

#include “SD.h”

file.read();

Parameters

File file

An instance of the File class (returned by SD.open())

Returns

The next byte (or character), or -1 if none is available.

Remarks

None

File::read()

Write data to the file.

Syntax

#include “SD.h”
file.write(data);
file.write(buf, len);

Parameters

File file
An instance of the File class (returned by SD.open())

type data
The data to print (type can be char, byte, int, long, or string)

char[] buf
An array of characters or bytes

int len
The number of elements in buf

Returns

write() will return the number of bytes written, though reading that number is optional.

Remarks

None

4. Introduction to SPI and 2-wireSlavesModes

4.1 SPI Slave

SPI circuit has following internal registers. How to use them will be described later.

3	2	1	0
MISC_REG	RBUF_STS_REG	WBUF_SET_REG	MAIN_REG
. . . . .			
RBUF_REG (3 ~ 0)			
RBUF_REG (7 ~ 4)			
RBUF_REG (11 ~ 8)			
RBUF_REG (15 ~ 12)			
WBUF_REG (3 ~ 0)			
WBUF_REG (7 ~ 4)			
WBUF_REG (11 ~ 8)			
WBUF_REG (15 ~ 12)			

RBUF related registers are to be read by the SPI host.

WBUF related registers are to be written by the SPI host.

MAIN_REG	
Bit#	Description
7	Indicate if a '1' was written to reset bit (bit-3) by remote SPI host. This bit is cleared by calling "*v8_spi_slave_reset_intr_clr()*" in "wiring_intr.c" and user may add his/her ISR to handle this event in "*SPI::isr()*" defined in "SPI.cpp".
6	Indicate if remote SPI host has completed reading data from RBUF_REG. This bit is cleared by calling "*v8_spi_slave_s2m_buffer_empty_intr_clr()*" in wiring_intr.c" and user can hook his/her ISR to handle this event by calling "*SPI::attachInterrupt*()" defined in "SIP.cpp".
5	Indicate if remote SPI host has completed writing data to WBUF_REG. This bit is cleared by calling "*v8_spi_slave_m2s_buffer_full_intr_clr()*" in wiring_intr.c" and user can hook his/her ISR to handle this event by calling "*SPI::attachInterrupt*()" defined in "SPI.cpp".
4	Reserved.
3	Remote host writes '1' to this bit to let SPI receiver enters reset mode, an interrupt will also be generated. This bit will automatically change to '0' when the reset action is done.
2	NavSpark writes'1' to this bit to indicate SPI host the RBUF_REG is ready to be read by calling "*SPI::enableBufferForHostRead()*". After host completes reading specified

	number of data, SPI receiver will trig an interrupt and set bit-6 to '1'. User can add his/her ISR by calling "*SPI::attachInterrupt()*" and also call "*SPI::disableBufferForHostRead()*" to notify SPI host no more data is valid.
1	NavSpark writes '1' to this bit to notify remote host the WBUF_REG is ready to be written by calling "*SPI::enableBufferForHostWrite()*", this bit will automatically clear to '0' when remote host writes specified number of data into WBUF_REG. User can add his/her ISR for this event by calling "*SPI::attachInterrupt()*".
0	Reserved.

WBUF_SET_REG	
Bit#	Description
7:0	SPI host writes the number of bytes of data to be written to WBUF_REG in following transaction.

RBUF_STS_REG	
Bit#	Description
7:0	NavSpark writes data to be read to RBUF_REG followed by writing the number of bytes of data in RBUF_REG to this register, the SPI host may read this register prior to issuing a buffer read.

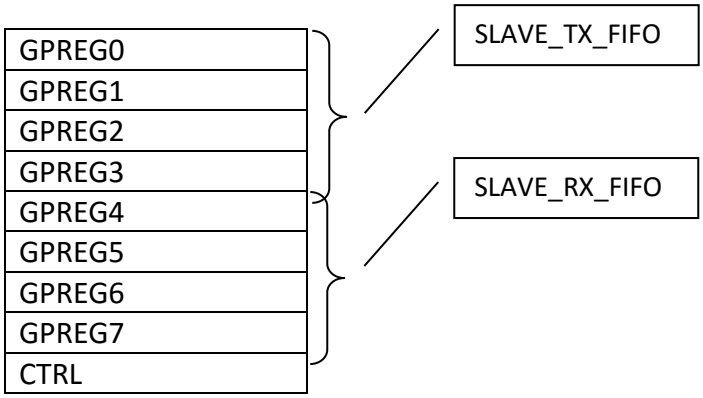
The SPI slave only accepts byte-aligned data and violation to this rule may cause unpredictable result and a reset command (issued by remote host) could be used to recover this error. Also the bit-order is MSB first in NavSpark, unchangeable. To access above registers from remote SPI host, the 1st byte of data is regarded as "command" and shown in below table.

Action of SPI host	Data on SPI bus
To write MAIN_REG	0x80 + 1-byte-data
To write WBUF_SET_REG	0x81 + 1-byte-data
To write RBUF_STS_REG	0x82 + 1-byte-data
To write MISC_REG	0x83 + 1-byte-data
To read MAIN_REG	0xC0 + 1-byte-data
To read WBUF_SET_REG	0xC1 + 1-byte-data
To read RBUF_STS_REG	0xC2 + 1-byte-data
To read MISC_REG	0xC3 + 1-byte-data
To write WBUF_REG	0x81 + 1-byte-count / 0x88 + N-byte-wdata (N<=16)
To read RBUF_REG	0xC2 + 1-byte-count / 0xC8 + N-byte-rdata (N<=16)

Normally remote SPI host communicates with NavSpark by write/read data to/from the WBUF/RBUF registers. See "demo_spi_master.ino" and "demo_spi_slave.ino" for more detail.

4.2 NavSpark as a 2-wire slave

Like SPI slave interface, NavSpark supports 2-wire slave interface and the 3rd party's ASIC could communicate with NavSpark by configuring itself as 2-wire host and NavSpark as slave. To gain the maximum flexibility NavSpark defines following internal registers and how to use them will be described later



There are 8 registers, GPREG0 ~ GPREG7,each 8-bit in width, used for data exchange between NavSpark and remote host. They are divided into two parts: the upper part is for NavSpark to write data to it and remote 2-wire host can read from it; the lower part is for NavSpark to read data written by remote 2-wire host. Currently NavSpark divides these registers into two equal parts and user can change the size of each part by modify the following parameters defined in "TwoWire.h".

```
#define TWOWIRE_SLAVE_TX_FIFO_BASE      0
#define TWOWIRE_SLAVE_TX_FIFO_SIZE      4
#define TWOWIRE_SLAVE_RX_FIFO_BASE      4
#define TWOWIRE_SLAVE_RX_FIFO_SIZE      4
```

The last register implemented for NavSpark as 2-wire slave is CTRL register which is used to exchange the control information between NavSpark and remote host.

CTRL	
Bit#	Description
7:5	NavSpark writes the number of data written to TX_FIFO in this field.
4;2	Host writes the number of data written to RX_FIFO in this field.
1	NavSpark writes '1' to this bit to notify host to read data from TX_FIFO, and host writes '0' to this bit to notify NavSpark can put new data to TX_FIFO. When this bit is '1' and host writes '0' to this bit, an internal interrupt will be trigged in NavSpark.

0	Host writes '1' to this bit to notify NavSpark to read data written in RX_FIFO, and NavSpark writes '0' to this bit to notify host to write new data to RX_FIFO. When host writes '1' to this bit, will trig an internal interrupt in NavSpark.

From view point of the 2-wire host, the register GPREG0 is addressed as offset 0 of NavSpark, GPREG1 is addressed as offset 1, and CTRL is addressed as offset 8. NavSpark has implemented an internal pointer to one of these 9 registers by which current read/write operation issued from host will be applied to, this pointer is forced to 0 after power-on reset. Basically the remote host can access NavSpark by following ways. Note that words in blue are driven by host and words in red are driven by NavSpark.

ST(1) + DEV_ADDR(7) + W(1) + ACK(1) + OFFSET(8) + ACK(1) + DATA(8) + ACK(1) + DATA(8) + … + ACK(1) + SP(1)
ST(1) + DEV_ADDR(7) + W(1) + ACK(1) + OFFSET(8) + ACK(1) + ST(1) + DEV_ADDR(7) + R(1) + DATA(8) + ACK(1) + DATA(8) + … + NACK(1) + SP(1)
ST(1) + DEV_ADDR(7) + R(1) + DATA(8) + ACK(1) + DATA(8) + … + NACK(1) + SP(1)

In the 1st case, the "DATA" will be written to the registers starting from the offset specified by "OFFSET" one by one. The NavSpark will auto-increment the internal pointer to next register after writing is done.

In the 2nd case, the 2-wire host informs NavSpark to move the internal pointer to registers specified by "OFFSET" and begin reading, NavSpark will respond putting"DATA" one by one to 2-wire bus.

In the 3rd case, the 2-wire host read the "DATA" directly from offset pointed by internal pointer.

5. Structure Reference

All the structures used in NavSpark can be found in the source files under directory "hardware\arduino\leon\cores\arduino". Most of them are put in "sti_gnss_lib.h".

6. Define Reference

All the defines used in NavSpark can be found in source files which are put in following directories.

- "hardware\arduino\leon\cores\arduino"
- "hardware\arduino\leon\variants\venus822\pins_arduino.h"

Change Log

Version 0.4, Dec 14, 2015

1. Add new members, SPIClass and TwoWire
2. Add new members, SDClass and File

Version 0.3, July 23, 2014

1. Change the altitude stored in GNSSAltitude to be MSL altitude which is same as output in NMEA.
2. Add a new member, GNSSDOP, in class GNSS.
3. Add a new member, GNSSGeoSeparation, in GNSS.

Version 0.2, May 22, 2014

1. PWM minimum frequency changed from 1Hz to 2Hz

Version 0.1, May 16, 2014

1. Initial release