XeThru Sensor Modules With Arduino

How to Use XeThru Sensor Modules With Arduino

XeThru Application Note **by Novelda AS**

Rev. A - August 28. 2018

Summary

This document describes how to connect a XeThru sensor module to Arduino. Valid for X4M200 and X4M300

1 Introduction

This document shows you how to use an Arduino to connect and communicate with the following modules:

- X4M200 (Respiration sensor)
- X4M300 (Presence sensor)

The idea behind this example is to show you how to set up and run a XeThru sensor module using serial communication. Each example are stand-alone examples that do not use any libraries.

The example code in this document was made for Arduino Mega because it has extra serial ports which makes it easier to debug. The code examples should work on any other Arduino if you change the *SerialDebug*-related code to some other way of outputting the result.

You can find the documentation for the serial communication protocol of X4-based modules in the XeThru Module Communication Protocol [1].

2 Connecting a XeThru Module to Arduino

2.1 Parts List

The following parts were used in this document:

- Breadboard
- Jumper wires
- XeThru Module (X4M200 or X4M300)
- Arduino Mega
- TXB0108 level shifter module [2]

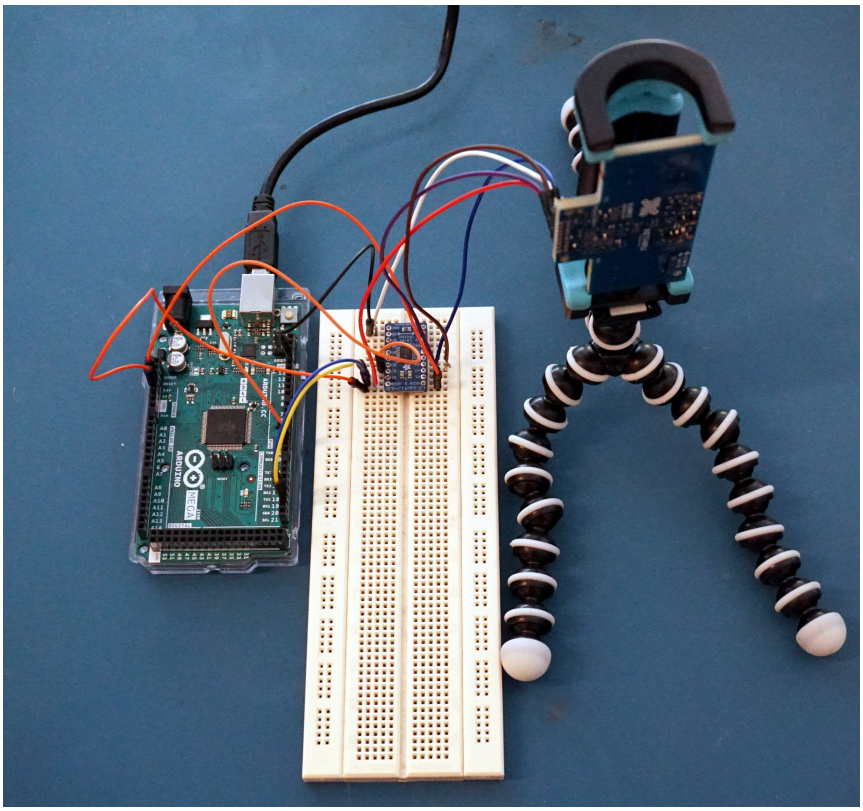

Fig 1. XeThru sensor connected to Arduino Mega.

2.2 Connection Diagram

To communicate between the Arduino and the XeThru module, use the modules RX and TX pins for serial connection (UART). And use a GPIO pin from the Arduino to control the *Reset* pin of the XeThru module. A level-shifter is needed to convert the Arduino Mega 5V logic to the 3.3V XeThru sensor voltage. The pins of the X4M200/X4M300 module are listed in section 16-pin XeThru Interface Connector below.

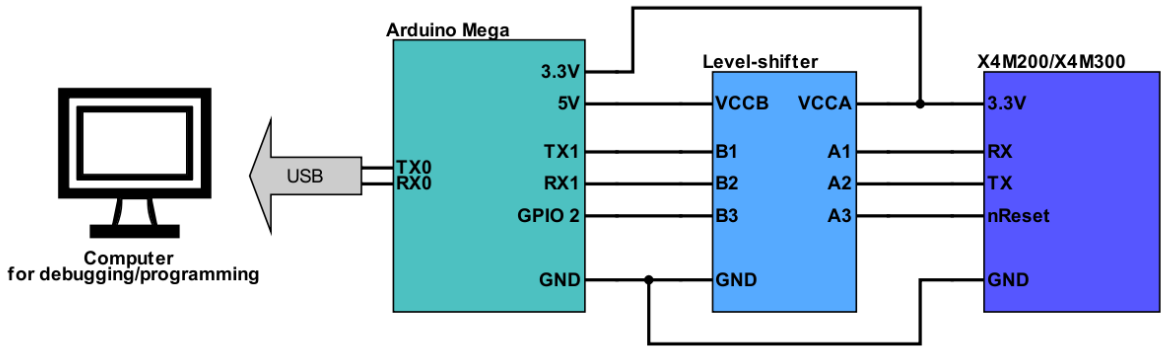

Fig.2 Connection Diagram

2.2.1 Reset Pin Control

If you hold the *RX* pin low during power-up or reset, the sensor modulewill go into bootloader mode to allow for firmware upgrades. You don't want this, so you need to make sure the *RX* pin is high when you power on the module. Unfortunately, it seems that manually setting the pin high in the Arduino code at startup is not done quick enough. The way to fix it is to control the *Reset* pin while making sure the *RX* pin of the radar is kept low while doing a reset after startup.

2.3 16-pin XeThru Interface Connector

The XeThru Modules have a 16-pin connector intended for interfacing a host board.

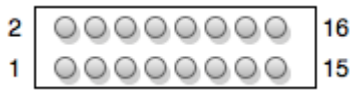

Fig. 3 XeThru 16-pin Connector

2.3.1 Pin Descriptions

Pin no	Description	Name	Type	Usage
1	Power, 3.3-5.5V	VDD_EXT	Power	Module power input
2	Power, GND	GND	Power	Module power input
3	USART RX / Force Bootloader	RX / BOOT	Input	USART receive / Holding pin 3 low during reset or power-up will force the unit into bootloader mode
4	USART TX	TX	Output	USART transmit
5	MODE SELECT 1 / USART SCK	MODESEL1	Input with pull-up	Mode select pin 1 / USART serial clock (USRT mode only)
6	MODE SELECT 2	MODESEL2	Input with pull-up	Mode select pin 2
7	Reset	nRESET	Input with pull-up	Active low module MCU reset
8	No Connect	N.C	N/A	Leave unconnected
9	No Connect	N.C	N/A	Leave unconnected
10	No Connect	N.C	N/A	Leave unconnected

Pin no	Description	Name	Type	Usage
11 to 16	IOx	IOx	I/O	Functionality is Profile specific

2.3.2 Electrical Specification

Parameter	Value	Comment
Supply Voltage VDD_EXT	3.3 - 5.5V	
IO-voltage range, nominal	-0.3 - 3.3V	
V_{IH} min	2.0V	Minimum input high threshold voltage
V_{IL} max	0.8V	Maximum input low threshold voltage

3 X4M200 Respiration Example

In this example you'll load the Adult Respiration Profile, configure a few settings, then start reading out respiration messages and printing out the RPM.

After loading the profile, it can be configured by sending profile configuration commands. After configuring the profile, you can send a command to start it. Next time the module is powered, it will automatically load the previous configuration and resume operation. If you want to change sensor behavior, stop the loaded profile, reconfigure and start again. Configuration workflow:

1. Set XeThru Sensor to stop mode
2. Load profile
3. Set parameters (detection zone, sensitivity, noise map, led, output, etc.)
4. Set XeThru Sensor to run mode to start profile
5. Read and parse messages from sensor to get RPM

Note: The sensor will only give out respiration for targets that are completely still, typically lying in a bed or sitting still, and breathing within 8 to 30 breaths per minute.

3.1 Arduino Code

You can also find latest version of this code example on GitHub: https://github.com/xethru/xethru-arduino/blob/master/X4M200_respiration_example/X4M200_respiration_example.ino

Arduino Code

```
// SERIAL PORTS:
// These definitions work for Arduino Mega, but must be changed for other Arduinos.
//  * Note: Using Serial as SerialRadar seems to give a few CRC errors. I'm not seeing this
//    using Serial1, Serial2, or Serial3. Could probably be solved by changing baud rate)
//
#define SerialRadar Serial1        // Used for communication with the radar
#define SerialDebug Serial        // Used for printing debug information
```

```
// Pin definitions
#define RESET_PIN 2
#define RADAR_RX_PIN 18


//
// The following values can be found in XeThru Module Communication Protocol:
//  https://www.xethru.com/community/resources/xethru-module-communication-protocol.130/
//
#define XT_START 0x7d
#define XT_STOP 0x7e
#define XT_ESCAPE 0x7f

#define XTS_ID_SLEEP_STATUS                             (uint32_t)0x2375a16c
#define XTS_ID_RESP_STATUS                              (uint32_t)0x2375fe26
#define XTS_ID_RESPIRATION_MOVINGLIST                   (uint32_t)0x610a3b00
#define XTS_ID_RESPIRATION_DETECTIONLIST                (uint32_t)0x610a3b02
#define XTS_ID_APP_RESPIRATION_2                        (uint32_t)0x064e57ad
#define XTS_ID_DETECTION_ZONE                           (uint32_t)0x96a10a1c
#define XTS_ID_SENSITIVITY                              (uint32_t)0x10a5112b

// Profile codes
#define XTS_VAL_RESP_STATE_BREATHING      0x00 // Valid RPM sensing
#define XTS_VAL_RESP_STATE_MOVEMENT       0x01 // Detects motion, but can not identify breath
#define XTS_VAL_RESP_STATE_MOVEMENT_TRACKING  0x02 // Detects motion, possible breathing soon
#define XTS_VAL_RESP_STATE_NO_MOVEMENT      0x03 // No movement detected
#define XTS_VAL_RESP_STATE_INITIALIZING     0x04 // Initializing sensor
#define XTS_VAL_RESP_STATE_ERROR        0x05 // Sensor has detected some problem. StatusValue
indicates problem.
#define XTS_VAL_RESP_STATE_UNKNOWN        0x06 // Undefined state.

#define XTS_SPR_APPDATA 0x50
#define XTS_SPR_SYSTEM 0x30

#define XTS_SPC_APPCOMMAND 0x10
#define XTS_SPC_MOD_SETMODE 0x20
#define XTS_SPC_MOD_LOADAPP 0x21
#define XTS_SPC_MOD_RESET 0x22
#define XTS_SPC_MOD_SETCOM 0x23
#define XTS_SPC_MOD_SETLEDCONTROL 0x24
#define XTS_SPC_MOD_NOISEMAP 0x25

// Output control
#define XTID_OUTPUT_CONTROL_DISABLE    (0)
#define XTID_OUTPUT_CONTROL_ENABLE     (1)

// Sensor mode IDs
#define XTS_SM_RUN                  (0x01)
#define XTS_SM_NORMAL               (0x10)
#define XTS_SM_IDLE                 (0x11)
#define XTS_SM_MANUAL               (0x12)
#define XTS_SM_STOP                 (0x13)

#define XTS_SPR_ACK 0x10
#define XTS_SPR_ERROR 0x20

#define XTS_SPRS_BOOTING (uint32_t)0x00000010
#define XTS_SPRS_READY (uint32_t)0x00000011
```

```
#define XTS_SPCA_SET 0x10
#define XTS_SPCN_SETCONTROL 0x10
#define XTS_SPCO_SETCONTROL 0x10
#define XTS_SPC_OUTPUT 0x41

#define TX_BUF_LENGTH 64
#define RX_BUF_LENGTH 64

unsigned char send_buf[TX_BUF_LENGTH];  // Buffer for sending data to radar.
unsigned char recv_buf[RX_BUF_LENGTH];  // Buffer for receiving data from radar.
const char * states[7] = { "Breathing", "Movement", "Movement tracking", "No movement", "Initiali
zing", "", "Unknown" };

// Struct to hold respiration message from radar
typedef struct RespirationMessage {
  uint32_t state_code;
  float rpm;
  float distance;
  uint32_t signal_quality;
  float movement_slow;
  float movement_fast;
};




void setup()
{
  // Getting the startup sequence right:
  //
  // If the RX-pin of the radar is low during reset or power-up, it goes into bootloader mode.
  // We don't want that, that's why we first set the RADAR_RX_PIN to high, then reset the module.
  pinMode(RADAR_RX_PIN, OUTPUT);
  digitalWrite(RADAR_RX_PIN, HIGH);
  pinMode(RESET_PIN, OUTPUT);
  digitalWrite(RESET_PIN, LOW);
  delay(100);
  digitalWrite(RESET_PIN, HIGH);

  // Set up serial communication
  SerialRadar.begin(115200);
  SerialDebug.begin(115200);

  // After the module resets, the XTS_SPRS_BOOTING message is sent. Then, after the
  // module booting sequence is completed and the module is ready to accept further
  // commands, the XTS_SPRS_READY command is issued. Let's wait for this.
  wait_for_ready_message();

  // Stop the module, in case it is running
  stop_module();

  // Load respiration profile
  load_profile(XTS_ID_APP_RESPIRATION_2);

  // Configure the noisemap
  configure_noisemap();

  // Set detection zone
  set_detection_zone(0.4, 2.0);
```

```
  // Set sensitivity
  set_sensitivity(9);

  // Enable only the Sleep message, disable all others
  enable_output_message(XTS_ID_SLEEP_STATUS);
  disable_output_message(XTS_ID_RESP_STATUS);
  disable_output_message(XTS_ID_RESPIRATION_MOVINGLIST);
  disable_output_message(XTS_ID_RESPIRATION_DETECTIONLIST);

  // Run profile - after this the radar will start sending the sleep message we enabled above
  run_profile();

}


void loop() {
  // For every loop we check to see if we have received any respiration data
  RespirationMessage msg;
  if (get_respiration_data(&msg)) {
    //Do something with msg...
    SerialDebug.print("State: ");
    SerialDebug.println(states[msg.state_code]);
    SerialDebug.print("RPM: ");
    SerialDebug.println(msg.rpm, 1);
    SerialDebug.print("Distance: ");
    SerialDebug.println(msg.distance, 1);
    SerialDebug.print("signal_quality: ");
    SerialDebug.println(msg.signal_quality);
    SerialDebug.print("movement_slow: ");
    SerialDebug.println(msg.movement_slow, 1);
    SerialDebug.print("movement_fast: ");
    SerialDebug.println(msg.movement_fast, 1);
    SerialDebug.println("---");
  }
}



int get_respiration_data(RespirationMessage * resp_msg) {

  // receive_data() fills recv_buf[] with data.
  if (receive_data() < 1)
    return 0;

  // Respiration Sleep message format:
  //
  // <Start> + <XTS_SPR_APPDATA> + [XTS_ID_SLEEP_STATUS(i)] + [Counter(i)]
  // + [StateCode(i)] + [RespirationsPerMinute(f)] + [Distance(f)]
  // + [SignalQuality(i)] + [MovementSlow(f)] + [MovementFast(f)]
  //

  // Check that it's a sleep message (XTS_ID_SLEEP_STATUS)
  uint32_t xts_id = *((uint32_t*)&recv_buf[2]);
  if (xts_id != XTS_ID_SLEEP_STATUS)
    return 0;

  // Extract the respiration message data:
  resp_msg->state_code = *((uint32_t*)&recv_buf[10]);
```

```
  resp_msg->rpm = *((float*)&recv_buf[14]);
  resp_msg->distance = *((float*)&recv_buf[18]);
  resp_msg->signal_quality = *((uint32_t*)&recv_buf[22]);
  resp_msg->movement_slow = *((float*)&recv_buf[26]);
  resp_msg->movement_fast = *((float*)&recv_buf[30]);

  // Return OK
  return 1;
}


// Stop module from running
void stop_module()
{
  // Empty the buffer before stopping the radar profile:
  while (SerialRadar.available())
    SerialRadar.read();

  // Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_SETMODE;
  send_buf[2] = XTS_SM_STOP;

  // Send the command
  send_command(3);

  // Get ACK response from radar
  get_ack();
}


// Set sensitivity
void set_sensitivity(uint32_t sensitivity)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_APPCOMMAND;
  send_buf[2] = XTS_SPCA_SET;
  send_buf[3] = XTS_ID_SENSITIVITY & 0xff;
  send_buf[4] = (XTS_ID_SENSITIVITY >> 8) & 0xff;
  send_buf[5] = (XTS_ID_SENSITIVITY >> 16) & 0xff;
  send_buf[6] = (XTS_ID_SENSITIVITY >> 24) & 0xff;
  send_buf[7] = sensitivity & 0xff;
  send_buf[8] = (sensitivity >> 8) & 0xff;
  send_buf[9] = (sensitivity >> 16) & 0xff;
  send_buf[10] = (sensitivity >> 24) & 0xff;

  //Send the command
  send_command(11);

  // Get ACK response from radar
  get_ack();
}



// Set detection zone
void set_detection_zone(float zone_start, float zone_end)
{
```

```
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_APPCOMMAND;
  send_buf[2] = XTS_SPCA_SET;
  send_buf[3] = XTS_ID_DETECTION_ZONE & 0xff;
  send_buf[4] = (XTS_ID_DETECTION_ZONE >> 8) & 0xff;
  send_buf[5] = (XTS_ID_DETECTION_ZONE >> 16) & 0xff;
  send_buf[6] = (XTS_ID_DETECTION_ZONE >> 24) & 0xff;

  // Copy the bytes of the floats to send buffer
  memcpy(send_buf+7, &zone_start, 4);
  memcpy(send_buf+11, &zone_end, 4);

  //Send the command
  send_command(15);

  // Get ACK response from radar
  get_ack();
}



// Run profile
void run_profile()
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_SETMODE;
  send_buf[2] = XTS_SM_RUN;

  //Send the command
  send_command(3);

  // Get ACK response from radar
  get_ack();
}


// Load profile
void load_profile(uint32_t profile)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_LOADAPP;
  send_buf[2] = profile & 0xff;
  send_buf[3] = (profile >> 8) & 0xff;
  send_buf[4] = (profile >> 16) & 0xff;
  send_buf[5] = (profile >> 24) & 0xff;

  //Send the command
  send_command(6);

  // Get ACK response from radar
  get_ack();
}

void configure_noisemap()
{
  // send_buf[3] Configuration:
```

```
  //
  // Bit 0: FORCE INITIALIZE NOISEMAP ON RESET
  // Bit 1: ADAPTIVE NOISEMAP ON
  // Bit 2: USE DEFAULT NOISEMAP
  //

  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_NOISEMAP;
  send_buf[2] = XTS_SPCN_SETCONTROL;
  send_buf[3] = 0x06; // 0x06: Use default noisemap and adaptive noisemap
  send_buf[4] = 0x00;
  send_buf[5] = 0x00;
  send_buf[6] = 0x00;

  //Send the command
  send_command(7);

  // Get ACK response from radar
  get_ack();
}


void enable_output_message(uint32_t message)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_OUTPUT;
  send_buf[2] = XTS_SPCO_SETCONTROL;
  send_buf[3] = message & 0xff;
  send_buf[4] = (message >> 8) & 0xff;
  send_buf[5] = (message >> 16) & 0xff;
  send_buf[6] = (message >> 24) & 0xff;
  send_buf[7] = XTID_OUTPUT_CONTROL_ENABLE & 0xff;
  send_buf[8] = (XTID_OUTPUT_CONTROL_ENABLE >> 8) & 0xff;
  send_buf[9] = (XTID_OUTPUT_CONTROL_ENABLE >> 16) & 0xff;
  send_buf[10] = (XTID_OUTPUT_CONTROL_ENABLE >> 24) & 0xff;

  //Send the command
  send_command(11);

  // Get ACK response from radar
  get_ack();
}


void disable_output_message(uint32_t message)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_OUTPUT;
  send_buf[2] = XTS_SPCO_SETCONTROL;
  send_buf[3] = message & 0xff;
  send_buf[4] = (message >> 8) & 0xff;
  send_buf[5] = (message >> 16) & 0xff;
  send_buf[6] = (message >> 24) & 0xff;
  send_buf[7] = XTID_OUTPUT_CONTROL_DISABLE & 0xff;
  send_buf[8] = (XTID_OUTPUT_CONTROL_DISABLE >> 8) & 0xff;
  send_buf[9] = (XTID_OUTPUT_CONTROL_DISABLE >> 16) & 0xff;
```

```
  send_buf[10] = (XTID_OUTPUT_CONTROL_DISABLE >> 24) & 0xff;

  //Send the command
  send_command(11);

  // Get ACK response from radar
  get_ack();
}



// This method waits indefinitely for the XTS_SPRS_READY message from the radar
void wait_for_ready_message()
{
  SerialDebug.println("Waiting for XTS_SPRS_READY...");
  while (true) {
    if (receive_data() < 1)
      continue;

    if (recv_buf[1] != XTS_SPR_SYSTEM)
      continue;

    //uint32_t response_code = (uint32_t)recv_buf[2] | ((uint32_t)recv_buf[3] << 8) | ((uint32_t)
recv_buf[4] << 16) | ((uint32_t)recv_buf[5] << 24);
    uint32_t response_code = *((uint32_t*)&recv_buf[2]);
    if (response_code == XTS_SPRS_READY) {
      SerialDebug.println("Received XTS_SPRS_READY!");
      return;
    }
    else if (response_code == XTS_SPRS_BOOTING)
      SerialDebug.println("Radar is booting...");
  }
}

// This method checks if an ACK was received from the radar
void get_ack()
{
  int len = receive_data();

  if (len == 0)
    SerialDebug.println("No response from radar");
  else if (len < 0)
    SerialDebug.println("Error in response from radar");
  else if (recv_buf[1] != XTS_SPR_ACK)  // Check response for ACK
    SerialDebug.println("Did not receive ACK!");
}


/*
 * Adds CRC, Escaping and Stop byte to the
 * send_buf and sends it over the SerialRadar.
 */
void send_command(int len)
{
  unsigned char crc = 0;

  // Calculate CRC
  for (int i = 0; i < len; i++)
```

```
    crc ^= send_buf[i];

  // Add CRC to send buffer
  send_buf[len] = crc;
  len++;

  // Go through send buffer and add escape characters where needed
  for (int i = 1; i < len; i++) {
    if (send_buf[i] == XT_START || send_buf[i] == XT_STOP || send_buf[i] == XT_ESCAPE)
    {
      // Shift following bytes one place up
      for (int u=len; u > i; u--)
        send_buf[u] = send_buf[u-1];

      // Add escape byte at old byte location
      send_buf[i] = XT_ESCAPE;

      // Increase length by one
      len++;

      // Increase counter so we don't process it again
      i++;
    }
  }

  // Send data (including CRC) and XT_STOP
  SerialRadar.write(send_buf, len);
  SerialRadar.write(XT_STOP);

  // Print out sent data for debugging:
  SerialDebug.print("Sent: ");
  for (int i = 0; i < len; i++) {
    SerialDebug.print(send_buf[i], HEX);
    SerialDebug.print(" ");
  }
  SerialDebug.println(XT_STOP, HEX);
}


/*
 * Receive data from radar module
 *  -Data is stored in the global array recv_buf[]
 *  -On success it returns number of bytes received (without escape bytes
 *  -On error it returns -1
 */
int receive_data() {

  int recv_len = 0;  //Number of bytes received

  // Wait 500 ms if nothing is available yet
  if (!SerialRadar.available())
    delay(500);

  // Wait for start character
  while (SerialRadar.available())
  {
    unsigned char c = SerialRadar.read();  // Get one byte from radar

    // If we receive an ESCAPE character, the next byte is never the real start character
```

```
  if (c == XT_ESCAPE)
  {
    // Wait for next byte and skip it.
    while (!SerialRadar.available());
    SerialRadar.read();
  }
  else if (c == XT_START)
  {
    // If it's the start character we fill the first character of the buffer and move on
    recv_buf[0] = c;
    recv_len = 1;
    break;
  }

  // Wait 10 ms if nothing is available yet
  if (!SerialRadar.available())
    delay(10);
}

// Wait 10 ms if nothing is available yet
if (!SerialRadar.available())
  delay(10);

// Start receiving the rest of the bytes
while (SerialRadar.available())
{
  // read a byte
  unsigned char c = SerialRadar.read(); // Get one byte from radar

  // is it an escape byte?
  if (c == XT_ESCAPE)
  {
    // If it's an escape character next character in buffer is data and not special character:
    while (!SerialRadar.available());
    c = SerialRadar.read();
  }
  // is it the stop byte?
  else if (c == XT_STOP) {
    // Fill response buffer, and increase counter
    recv_buf[recv_len++] = c;
    break;  //Exit this loop
  }

  if (recv_len >= RX_BUF_LENGTH) {
    SerialDebug.println("BUFFER OVERFLOW!");
    return -1;
  }

  // Fill response buffer, and increase counter
  recv_buf[recv_len++] = c;



  // Wait 10 ms if nothing is available yet
  if (!SerialRadar.available())
    delay(10);
}

// Print received data
```

```
  #if 0
  SerialDebug.print("Received: ");
  for (int i = 0; i < recv_len; i++) {
    SerialDebug.print(recv_buf[i], HEX);
    SerialDebug.print(" ");
  }
  SerialDebug.println(" ");
  #endif

  // If nothing was received, return 0.
  if (recv_len==0)
    return 0;

  // If stop character was not received, return with error.
  if (recv_buf[recv_len-1] != XT_STOP)
    return -1;


  //
  // Calculate CRC
  //
  unsigned char crc = 0;

  // CRC is calculated without the crc itself and the stop byte, hence the -2 in the counter
  for (int i = 0; i < recv_len-2; i++)
    crc ^= recv_buf[i];

  // Check if calculated CRC matches the recieved
  if (crc == recv_buf[recv_len-2])
  {
    return recv_len;  // Return length of received data (without escape bytes) upon success
  }
  else
  {
    SerialDebug.print("CRC mismatch: ");
    SerialDebug.print(crc, HEX);
    SerialDebug.print(" != ");
    SerialDebug.println(recv_buf[recv_len-2], HEX);
    return -1; // Return -1 upon crc failure
  }
}
```

4 X4M300 Presence Example Code

In this example you'll load the Presence Profile, configure a few settings, then start reading out presence messages and printing out the presence state.

After loading the profile, it can be configured by sending profile configuration commands. After configuring the profile, you can send a command to start it. Next time the module is powered, it will automatically load the previous configuration and resume operation. If you want to change sensor behavior, stop the loaded profile, reconfigure and start again. Configuration workflow:

1. Set XeThru Sensor to stop mode
2. Load profile

3. Set parameters (detection zone, sensitivity, noise map, led, output, etc.)
4. Set XeThru Sensor to run mode to start profile
5. Read and parse messages from sensor to get presence state

4.1 Arduino Code

You can also find the latest version of this code example on GitHub: https://github.com/xethru/xethru-arduino/blob/master/X4M300_presence_example/X4M300_presence_example.ino

Arduino Code

```
// SERIAL PORTS:
// These definitions work for Arduino Mega, but must be changed for other Arduinos.
//  * Note: Using Serial as SerialRadar seems to give a few CRC errors. I'm not seeing this
//    using Serial1, Serial2, or Serial3. Could probably be solved by changing baud rate)
//
#define SerialRadar Serial1    // Used for communication with the radar
#define SerialDebug Serial    // Used for printing debug information

// Pin definitions
#define RESET_PIN 2
#define RADAR_RX_PIN 18


//
// The following values can be found in XeThru Module Communication Protocol:
//  https://www.xethru.com/community/resources/xethru-module-communication-protocol.130/
//
#define XT_START 0x7d
#define XT_STOP 0x7e
#define XT_ESCAPE 0x7f

#define XTS_ID_PRESENCE_SINGLE                          (uint32_t)0x723bfa1e
#define XTS_ID_PRESENCE_MOVINGLIST                      (uint32_t)0x723bfa1f

#define XTS_ID_APP_PRESENCE_2                           (uint32_t)0x014d4ab8
#define XTS_ID_DETECTION_ZONE                           (uint32_t)0x96a10a1c
#define XTS_ID_SENSITIVITY                              (uint32_t)0x10a5112b

// Profile codes
#define XTS_VAL_PRESENCE_PRESENCESTATE_NO_PRESENCE      0x00 // No presence detected
#define XTS_VAL_PRESENCE_PRESENCESTATE_PRESENCE         0x01 // Presence detect
#define XTS_VAL_PRESENCE_PRESENCESTATE_INITIALIZING     0x02 // Initializing
#define XTS_VAL_PRESENCE_PRESENCESTATE_UNKNOWN          0x03 // Unknown


#define XTS_SPR_APPDATA 0x50
#define XTS_SPR_SYSTEM 0x30

#define XTS_SPC_APPCOMMAND 0x10
#define XTS_SPC_MOD_SETMODE 0x20
#define XTS_SPC_MOD_LOADAPP 0x21
#define XTS_SPC_MOD_RESET 0x22
#define XTS_SPC_MOD_SETCOM 0x23
#define XTS_SPC_MOD_SETLEDCONTROL 0x24
#define XTS_SPC_MOD_NOISEMAP 0x25
```

```
// Output control
#define XTID_OUTPUT_CONTROL_DISABLE    (0)
#define XTID_OUTPUT_CONTROL_ENABLE     (1)

// Sensor mode IDs
#define XTS_SM_RUN                  (0x01)
#define XTS_SM_NORMAL               (0x10)
#define XTS_SM_IDLE                 (0x11)
#define XTS_SM_MANUAL               (0x12)
#define XTS_SM_STOP                 (0x13)

#define XTS_SPR_ACK 0x10
#define XTS_SPR_ERROR 0x20

#define XTS_SPRS_BOOTING  (uint32_t)0x00000010
#define XTS_SPRS_READY    (uint32_t)0x00000011

#define XTS_SPCA_SET 0x10
#define XTS_SPCN_SETCONTROL 0x10
#define XTS_SPCO_SETCONTROL 0x10
#define XTS_SPC_OUTPUT 0x41

#define TX_BUF_LENGTH 64
#define RX_BUF_LENGTH 64

unsigned char send_buf[TX_BUF_LENGTH];  // Buffer for sending data to radar.
unsigned char recv_buf[RX_BUF_LENGTH];  // Buffer for receiving data from radar.
const char * states[4] = { "No Presence", "Presence", "Initializing", "Unknown" };

// Struct to hold respiration message from radar
typedef struct PresenceMessage {
  uint32_t state_code;
  float distance;
  uint8_t dir;
  uint32_t signal_quality;
};


void setup()
{
  // Getting the startup sequence right:
  //
  // If the RX-pin of the radar is low during reset or power-up, it goes into bootloader mode.
  // We don't want that, that's why we first set the RADAR_RX_PIN to high, then reset the module.
  pinMode(RADAR_RX_PIN, OUTPUT);
  digitalWrite(RADAR_RX_PIN, HIGH);
  pinMode(RESET_PIN, OUTPUT);
  digitalWrite(RESET_PIN, LOW);
  delay(100);
  digitalWrite(RESET_PIN, HIGH);

  // Set up serial communication
  SerialRadar.begin(115200);
  SerialDebug.begin(115200);

  // After the module resets, the XTS_SPRS_BOOTING message is sent. Then, after the
  // module booting sequence is completed and the module is ready to accept further
  // commands, the XTS_SPRS_READY command is issued. Let's wait for this.
  wait_for_ready_message();
```

```
  // Stop the module, in case it is running
  stop_module();

  // Load Presence profile
  load_profile(XTS_ID_APP_PRESENCE_2);

  // Configure the noisemap
  configure_noisemap();

  // Set detection zone
  set_detection_zone(0.4, 1.0);

  // Set sensitivity
  set_sensitivity(9);

  // Enable only the Presence message, disable all others
  enable_output_message(XTS_ID_PRESENCE_SINGLE);
  disable_output_message(XTS_ID_PRESENCE_MOVINGLIST);

  // Run profile - after this the radar will start sending the sleep message we enabled above
  run_profile();

}


void loop() {
  // For every loop we check to see if we have received any presence data
  PresenceMessage msg;
  if (get_presence_data(&msg)) {
    //Do something with msg...
    SerialDebug.print("State: ");
    SerialDebug.println(states[msg.state_code]);
    SerialDebug.print("Distance: ");
    SerialDebug.println(msg.distance, 2);
    SerialDebug.print("Direction: ");
    SerialDebug.println(msg.dir);
    SerialDebug.print("signal_quality: ");
    SerialDebug.println(msg.signal_quality);
    SerialDebug.println("---");
  }
}


int get_presence_data(PresenceMessage * pres_msg) {

  // receive_data() fills recv_buf[] with data.
  if (receive_data() < 1)
    return 0;

  // Presence message format:
  //
  // <Start> + <XTS_SPR_APPDATA> + [XTS_ID_PRESENCE_SINGLE(i)] + [Counter(i)]
  // + [PresenceState(i)] + [Distance(f)] + <Direction> + [SignalQuality(i)] + <CRC>
  // + <End>
  //

  // Check that it's a presence message (XTS_ID_PRESENCE_SINGLE)
  uint32_t xts_id = *((uint32_t*)&recv_buf[2]);
```

```
  if (xts_id != XTS_ID_PRESENCE_SINGLE)
    return 0;

  // Extract the respiration message data:
  pres_msg->state_code = *((uint32_t*)&recv_buf[10]);
  pres_msg->distance = *((float*)&recv_buf[14]);
  pres_msg->dir = *((uint8_t*)&recv_buf[18]);
  pres_msg->signal_quality = *((uint32_t*)&recv_buf[19]);

  // Return OK
  return 1;
}



// Stop module from running
void stop_module()
{
  // Empty the buffer before stopping the radar profile:
  while (SerialRadar.available())
    SerialRadar.read();

  // Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_SETMODE;
  send_buf[2] = XTS_SM_STOP;

  // Send the command
  send_command(3);

  // Get ACK response from radar
  get_ack();
}


// Set sensitivity
void set_sensitivity(uint32_t sensitivity)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_APPCOMMAND;
  send_buf[2] = XTS_SPCA_SET;
  send_buf[3] = XTS_ID_SENSITIVITY & 0xff;
  send_buf[4] = (XTS_ID_SENSITIVITY >> 8) & 0xff;
  send_buf[5] = (XTS_ID_SENSITIVITY >> 16) & 0xff;
  send_buf[6] = (XTS_ID_SENSITIVITY >> 24) & 0xff;
  send_buf[7] = sensitivity & 0xff;
  send_buf[8] = (sensitivity >> 8) & 0xff;
  send_buf[9] = (sensitivity >> 16) & 0xff;
  send_buf[10] = (sensitivity >> 24) & 0xff;

  //Send the command
  send_command(11);

  // Get ACK response from radar
  get_ack();
}
```

```
// Set detection zone
void set_detection_zone(float zone_start, float zone_end)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_APPCOMMAND;
  send_buf[2] = XTS_SPCA_SET;
  send_buf[3] = XTS_ID_DETECTION_ZONE & 0xff;
  send_buf[4] = (XTS_ID_DETECTION_ZONE >> 8) & 0xff;
  send_buf[5] = (XTS_ID_DETECTION_ZONE >> 16) & 0xff;
  send_buf[6] = (XTS_ID_DETECTION_ZONE >> 24) & 0xff;

  // Copy the bytes of the floats to send buffer
  memcpy(send_buf+7, &zone_start, 4);
  memcpy(send_buf+11, &zone_end, 4);

  //Send the command
  send_command(15);

  // Get ACK response from radar
  get_ack();
}



// Run profile
void run_profile()
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_SETMODE;
  send_buf[2] = XTS_SM_RUN;

  //Send the command
  send_command(3);

  // Get ACK response from radar
  get_ack();
}


// Load profile
void load_profile(uint32_t profile)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_LOADAPP;
  send_buf[2] = profile & 0xff;
  send_buf[3] = (profile >> 8) & 0xff;
  send_buf[4] = (profile >> 16) & 0xff;
  send_buf[5] = (profile >> 24) & 0xff;

  //Send the command
  send_command(6);

  // Get ACK response from radar
  get_ack();
}
```

```
void configure_noisemap()
{
  // send_buf[3] Configuration:
  //
  // Bit 0: FORCE INITIALIZE NOISEMAP ON RESET
  // Bit 1: ADAPTIVE NOISEMAP ON
  // Bit 2: USE DEFAULT NOISEMAP
  //

  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_MOD_NOISEMAP;
  send_buf[2] = XTS_SPCN_SETCONTROL;
  send_buf[3] = 0x06; // 0x06: Use default noisemap and adaptive noisemap
  send_buf[4] = 0x00;
  send_buf[5] = 0x00;
  send_buf[6] = 0x00;

  //Send the command
  send_command(7);

  // Get ACK response from radar
  get_ack();
}


void enable_output_message(uint32_t message)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_OUTPUT;
  send_buf[2] = XTS_SPCO_SETCONTROL;
  send_buf[3] = message & 0xff;
  send_buf[4] = (message >> 8) & 0xff;
  send_buf[5] = (message >> 16) & 0xff;
  send_buf[6] = (message >> 24) & 0xff;
  send_buf[7] = XTID_OUTPUT_CONTROL_ENABLE & 0xff;
  send_buf[8] = (XTID_OUTPUT_CONTROL_ENABLE >> 8) & 0xff;
  send_buf[9] = (XTID_OUTPUT_CONTROL_ENABLE >> 16) & 0xff;
  send_buf[10] = (XTID_OUTPUT_CONTROL_ENABLE >> 24) & 0xff;

  //Send the command
  send_command(11);

  // Get ACK response from radar
  get_ack();
}


void disable_output_message(uint32_t message)
{
  //Fill send buffer
  send_buf[0] = XT_START;
  send_buf[1] = XTS_SPC_OUTPUT;
  send_buf[2] = XTS_SPCO_SETCONTROL;
  send_buf[3] = message & 0xff;
  send_buf[4] = (message >> 8) & 0xff;
  send_buf[5] = (message >> 16) & 0xff;
```

```
  send_buf[6] = (message >> 24) & 0xff;
  send_buf[7] = XTID_OUTPUT_CONTROL_DISABLE & 0xff;
  send_buf[8] = (XTID_OUTPUT_CONTROL_DISABLE >> 8) & 0xff;
  send_buf[9] = (XTID_OUTPUT_CONTROL_DISABLE >> 16) & 0xff;
  send_buf[10] = (XTID_OUTPUT_CONTROL_DISABLE >> 24) & 0xff;

  //Send the command
  send_command(11);

  // Get ACK response from radar
  get_ack();
}



// This method waits indefinitely for the XTS_SPRS_READY message from the radar
void wait_for_ready_message()
{
  SerialDebug.println("Waiting for XTS_SPRS_READY...");
  while (true) {
    if (receive_data() < 1)
      continue;

    if (recv_buf[1] != XTS_SPR_SYSTEM)
      continue;

    uint32_t response_code = (uint32_t)recv_buf[2] | ((uint32_t)recv_buf[3] << 8) | ((uint32_t)
recv_buf[4] << 16) | ((uint32_t)recv_buf[5] << 24);
    if (response_code == (uint32_t)XTS_SPRS_READY) {
      SerialDebug.println("Received XTS_SPRS_READY!");
      return;
    }
    else if (response_code == (uint32_t)XTS_SPRS_BOOTING)
      SerialDebug.println("Radar is booting...");
  }
}

// This method checks if an ACK was received from the radar
void get_ack()
{
  int len = receive_data();

  if (len == 0)
    SerialDebug.println("No response from radar");
  else if (len < 0)
    SerialDebug.println("Error in response from radar");
  else if (recv_buf[1] != XTS_SPR_ACK)  // Check response for ACK
    SerialDebug.println("Did not receive ACK!");
}



/*
 * Adds CRC, Escaping and Stop byte to the
 * send_buf and sends it over the SerialRadar.
 */
void send_command(int len)
{
  unsigned char crc = 0;
```

```
  // Calculate CRC
  for (int i = 0; i < len; i++)
    crc ^= send_buf[i];

  // Add CRC to send buffer
  send_buf[len] = crc;
  len++;

  // Go through send buffer and add escape characters where needed
  for (int i = 1; i < len; i++) {
    if (send_buf[i] == XT_START || send_buf[i] == XT_STOP || send_buf[i] == XT_ESCAPE)
    {
      // Shift following bytes one place up
      for (int u=len; u > i; u--)
        send_buf[u] = send_buf[u-1];

      // Add escape byte at old byte location
      send_buf[i] = XT_ESCAPE;

      // Increase length by one
      len++;

      // Increase counter so we don't process it again
      i++;
    }
  }

  // Send data (including CRC) and XT_STOP
  SerialRadar.write(send_buf, len);
  SerialRadar.write(XT_STOP);

  // Print out sent data for debugging:
  SerialDebug.print("Sent: ");
  for (int i = 0; i < len; i++) {
    SerialDebug.print(send_buf[i], HEX);
    SerialDebug.print(" ");
  }
  SerialDebug.println(XT_STOP, HEX);
}


/*
 * Receive data from radar module
 *  -Data is stored in the global array recv_buf[]
 *  -On success it returns number of bytes received (without escape bytes
 *  -On error it returns -1
 */
int receive_data() {

  int recv_len = 0;  //Number of bytes received

  // Wait 500 ms if nothing is available yet
  if (!SerialRadar.available())
    delay(500);

  // Wait for start character
  while (SerialRadar.available())
  {
```

```
  unsigned char c = SerialRadar.read();  // Get one byte from radar

  // If we receive an ESCAPE character, the next byte is never the real start character
  if (c == XT_ESCAPE)
  {
    // Wait for next byte and skip it.
    while (!SerialRadar.available());
    SerialRadar.read();
  }
  else if (c == XT_START)
  {
    // If it's the start character we fill the first character of the buffer and move on
    recv_buf[0] = c;
    recv_len = 1;
    break;
  }

  // Wait 10 ms if nothing is available yet
  if (!SerialRadar.available())
    delay(10);
}

// Wait 10 ms if nothing is available yet
if (!SerialRadar.available())
  delay(10);

// Start receiving the rest of the bytes
while (SerialRadar.available())
{
  // read a byte
  unsigned char c = SerialRadar.read(); // Get one byte from radar

  // is it an escape byte?
  if (c == XT_ESCAPE)
  {
    // If it's an escape character next character in buffer is data and not special character:
    while (!SerialRadar.available());
    c = SerialRadar.read();
  }
  // is it the stop byte?
  else if (c == XT_STOP) {
    // Fill response buffer, and increase counter
    recv_buf[recv_len++] = c;
    break;  //Exit this loop
  }

  if (recv_len >= RX_BUF_LENGTH) {
    SerialDebug.println("BUFFER OVERFLOW!");
    return -1;
  }

  // Fill response buffer, and increase counter
  recv_buf[recv_len++] = c;



  // Wait 10 ms if nothing is available yet
  if (!SerialRadar.available())
    delay(10);
```

```
  }

  // Print received data
  #if 0
  SerialDebug.print("Received: ");
  for (int i = 0; i < recv_len; i++) {
    SerialDebug.print(recv_buf[i], HEX);
    SerialDebug.print(" ");
  }
  SerialDebug.println(" ");
  #endif

  // If nothing was received, return 0.
  if (recv_len==0)
    return 0;

  // If stop character was not received, return with error.
  if (recv_buf[recv_len-1] != XT_STOP)
    return -1;


  //
  // Calculate CRC
  //
  unsigned char crc = 0;

  // CRC is calculated without the crc itself and the stop byte, hence the -2 in the counter
  for (int i = 0; i < recv_len-2; i++)
    crc ^= recv_buf[i];

  // Check if calculated CRC matches the recieved
  if (crc == recv_buf[recv_len-2])
  {
    return recv_len;  // Return length of received data (without escape bytes) upon success
  }
  else
  {
    SerialDebug.print("CRC mismatch: ");
    SerialDebug.print(crc, HEX);
    SerialDebug.print(" != ");
    SerialDebug.println(recv_buf[recv_len-2], HEX);
    return -1; // Return -1 upon crc failure
  }
}
```

5 References

[1]	Novelda, "XeThru Module Communication Protocol", https://www.xethru.com/community/resources/xethru-module-communication-protocol.130/
[2]	Adafruit, "Logic Level Converter", https://www.adafruit.com/product/395

6 Document History

Rev.	Release date	Change description
A	28-Aug-2018	Initial Release

7 Disclaimer

The information in this document is provided in connection with Novelda products. No license, express or implied, by estoppel or otherwise, to any intellectual property right is granted by this document or in connection with the sale of Novelda products. EXCEPT AS SET FORTH IN THE NOVELDA TERMS AND CONDITIONS OF SALES LOCATED ON THE NOVELDA WEBSITE, NOVELDA ASSUMES NO LIABILITY WHATSOEVER AND DISCLAIMS ANY EXPRESS, IMPLIED OR STATUTORY WARRANTY RELATING TO ITS PRODUCTS INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. IN NO EVENT SHALL NOVELDA BE LIABLE FOR ANY DIRECT, INDIRECT, CONSEQUENTIAL, PUNITIVE, SPECIAL OR INCIDENTAL DAMAGES (INCLUDING, WITHOUT LIMITATION, DAMAGES FOR LOSS AND PROFITS, BUSINESS INTERRUPTION, OR LOSS OF INFORMATION) ARISING OUT OF THE USE OR INABILITY TO USE THIS DOCUMENT, EVEN IF NOVELDA HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Novelda makes no representations or warranties with respect to the accuracy or completeness of the contents of this document and reserves the right to make changes to specifications and products descriptions at any time without notice. Novelda does not make any commitment to update the information contained herein. Unless specifically provided otherwise, Novelda products are not suitable for, and shall not be used in, automotive applications. Novelda products are not intended, authorized, or warranted for use as components in applications intended to support or sustain life.