AIR Base-LINK

Programmers Guide

March 4, 2013

TABLE OF CONTENTS

1. Introduction 4
1.1. Getting Started 4
1.1.1. Features 4
1.1.2. Required Peripheral Support 5
1.1.3. Applications 5
1.2. Terminology 5
1.2.1. Node Types 5
1.2.2. Data Transfer 6
1.2.3. Data Types 6
1.3. Architecture Overview 7
1.3.1. Physical Layer 8
1.3.2. Data Link Layer 8
1.4. Regulatory Compliance 10
1.5. Software Licensing Agreement 10
2. Hardware Interface 10
2.1. Hardware Abstraction Layer 10
3. Configuration 11
3.1. Microcontroller Global Interrupt Control 11
3.2. Platform Characteristics 12
3.3. Radio Characteristics 12
3.4. Protocol Characteristics 13
4. Basics Tutorial 13
4.1. Initial Setup 14
4.2. Gateway Node 14
4.3. Simplex End Point Node 15
4.4. Half Duplex End Point Node 16
5. Application Programming Interface Reference 16
5.1. Structures 16
5.1.1. Setup Information 16
5.1.2. Status Information 17
5.2. Functions 18
5.2.1. Configuration and Status 18

5.2.2. Connection Management 19
5.2.3. Data Transfers 20
5.2.4. Interrupt Service Routines 22
Appendix A. Example Usage 23
Simplex Communication 23
End Point Node(s) 24
Gateway Node 27
Appendix B. Platform Port 30
Appendix C. IAR Embedded Workbench (EW430) Project Setup 34
Document History 37

1. INTRODUCTION

This document discusses an open-source lightweight asynchronous wireless sensor network (WSN) protocol. It offers an easily configurable solution to many common wireless sensor network problems. The protocol can be used in a variety of wireless sensor network applications.

1.1. GETTING STARTED

The protocol provides an Application Programming Interface and configuration options to easily create a network. The protocol works with Anaren's AIR modules. The supported modules are listed below:

- Anaren AIR Proprietary RF Modules (A1101R04x, A1101R08x, A1101R09x, A110LR09x, and A2500R24x)

Note More modules may be supported at a later time.

The protocol features a simple interface to port between hardware platforms. Please refer to Hardware Abstraction Layer for more information.

1.1.1. FEATURES

To provide a lightweight wireless sensor network solution, many of the protocol features can be enabled or disabled. This allows complete flexibility on the application development end. A network can now be developed around the hardware available to it. Less capable platforms that are meant for status can use minimal protocol features while more powerful platforms can take advantage of the intelligent features that the protocol offers.
The protocol **provides** the application developer with the following features:

- Configurable
- Platform agnostic
- Small memory footprint
- Scalable
- Event-driven data transfers
- Low power consumption
- Single channel communication

The protocol **does not provide** the following:

- Frequency agility or hopping
- Collision avoidance (listen-before-talk)
- Encryption
- Duty cycling

The protocol meets certification requirements for Anaren Integrated Radio (AIR) modules as long as duty cycle requirements are met by the application (if duty cycling is required).

1.1.2. REQUIRED PERIPHERAL SUPPORT

To operate the supported radio hardware and protocol, it is required that the target platform has a 3-wire Serial Interface (SPI), hardware timer to provide a tick, available General Purpose Input/Output (GPIO) pins, and GPIO interrupt support.
The radio requires two GPIO pins. One GPIO pin is used in conjunction with the 3-wire SPI to provide a chip select (CSn) signal. The second GPIO pin is used for radio interrupts and is known as a General Digital Output (GDO). The specific GDO pin used for this protocol is GDO0. The following table describes the hardware dependencies and their usage in the protocol:

SPI (3-Wire)	**GPIO (CSn)**	**GPIO (GDO0)**
Communication channel between the MCU and the RF transceiver. This can be supported by a dedicated hardware peripheral or by bit banging.	SPI bus RF transceiver device selection. When asserted low, the RF transceiver can be communicated with.	General Digital Output of the RF transceiver. This pin is used for interrupts from the radio on SYNC word and End-of-Packet (EOP). This occurs for both receive and transmit radio operations.

1.1.3. APPLICATIONS

The protocol has been designed for and targets the following types of applications:

- Data logging
- Heating, ventilation, and air conditioning (HVAC)
- Home automation and control
- Intelligent occupancy sensing

1.2. TERMINOLOGY

1.2.1. NODE TYPES

GATEWAY

A Gateway is the sink for all network data. There is only one Gateway per network. A Gateway may also be used to bridge two different networks. The Gateway is a passive node in the network – it waits to be polled.

END POINT

An End Point acts as an initiator for all network activity. End Points dictate when data is transmitted to the Gateway. Since this protocol is event-driven, the End Point produces all timing characteristics of the network. An End Point is an active node in the network – it actively seeks access to the communication channel.

1.2.2. DATA TRANSFER

DATA TRANSFER

A data transfer refers to data traversing the network. The data can transverse one-way (simplex) or two-ways (half-duplex).

SIMPLEX

A simplex data transfer is a simple one-way communication path over the channel. Data is sent from node A to node B always. This is usually ***performed using a broadcast address*** as simplex data transfers do not perform a connection handshake; addressing must be broadcast (no addressing) or fixed at compile-time.

HALF-DUPLEX

A half-duplex data transfer is a two-way communication path over the channel. Data is sent from node A to node B and from node B back to node A.

1.2.3. DATA TYPES

MESSAGE

Message is used to generalize the encapsulation of the data being transmitted from node A to node B. It is to be interpreted as being completely independent of the layer with which it is being associated. When data is being mentioned as a message it means that data is being transmitted from node A to node B and is completely independent of any header and footer dependencies that other data types might suggest.
Message is also used to refer to the protocol subscription service.
Due to its dual reference, it is important to note the context in which the term message is used.

DATA STREAM

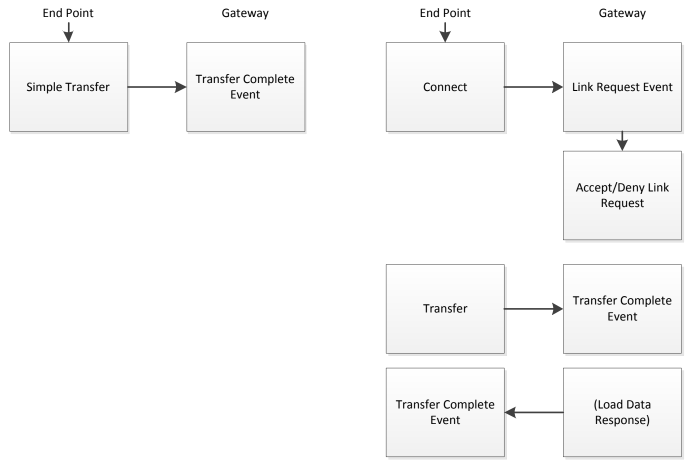
A data stream, also known as a bit stream, refers to data at the Physical layer. A data stream can be made up of an optional header and footer. The Physical layer defines how the data stream is modulated (i.e. OOK, 2-FSK, etc.).

FRAME

A frame refers to data at the Data Link layer. A frame is made up of header information and optional footer information. Frames are used to encapsulate messages to send to another node that contains the same Data Link layer. Data Link layers will pack and unpack all frames.

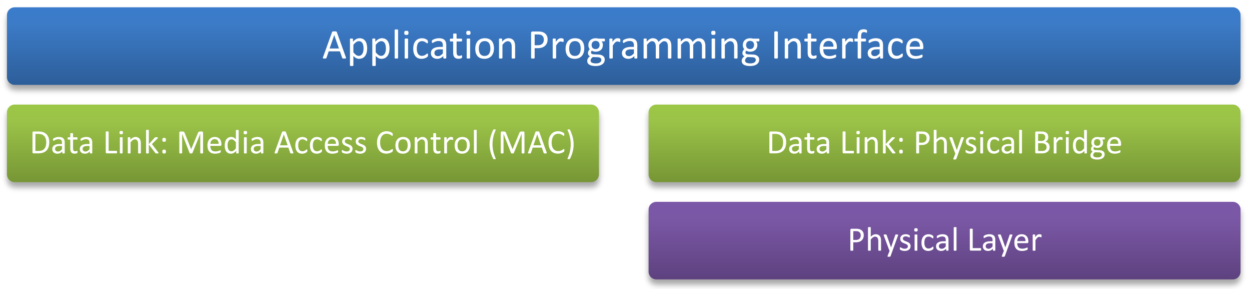
1.3. TYPICAL OPERATION

All transfers begin with an End Point node – the Gateway node always begins in a listening state. When performing simple transfers, no connection is required. When performing two-way transfers, the End Point node must connect first before performing any transfers.



1.4. ARCHITECTURE OVERVIEW

The protocol is made up of two main layers. They are the Physical and Data Link layers. Each layer has a specific role that is based on the well-known Open Systems Interconnection (OSI) model. An Application Programming Interface (API) sits on top of the OSI layers.
The application developer will not interact with these layers directly. Instead, the application developer will interact with the Application Programming Interface (API). The API provides the interface for controlling the protocol.
The following diagram describes the high level protocol design:


Application Programming Interface
Data Link: Media Access Control (MAC)
Data Link: Physical Bridge
Physical Layer

The layers below the API work amongst one another to get data from point A to point B. Application developers should use the API to make direct calls into the protocol; there is no need to modify any of the protocol layers.

1.4.1. PHYSICAL LAYER

The Physical layer is made up of an Anaren Integrated Radio (AIR) module that is based on the Texas Instruments (TI) CC1101, CC110L, and CC2500 chipset and a hardware timer. The timer is used for events within the protocol such as handling timeouts.
The physical layer **must be** implemented using Anaren's CC1101-based AIR module software interfaces. These interfaces contain all required certification information for that module. ***If the physical layer is not implemented using the AIR module interface, certification may no longer be valid.*** The following list contains all CC1101-based AIR module interfaces: A1101R04x, A1101R08x, A1101R09x, A110LR09x, and A2500R24x.
It is recommended that the Anaren CC110x2500 device driver be used in conjunction with the AIR module interface. This driver requires a SPI and GDO0 interrupt interface. These two pieces make up the majority of the Physical layer.
The protocol can be adapted to any hardware platform that meets its basic requirements. A hardware platform is supported if it provides the following:

RF Transceiver	Timer	I/O
▪ Anaren Integrated Radio (AIR) module with a TI CC1101, CC110L, or CC2500 chipset only	▪ Configured to a 1ms tick period (longer tick periods are valid but may reduce protocol performance)	▪ One dedicated 3-wire Serial Peripheral Interface (SPI) ▪ One dedicated GPIO for SPI chip select (CSn) ▪ One dedicated GPIO for radio interrupts

1.4.2. DATA LINK LAYER

PHYSICAL BRIDGE

The *Physical Bridge* provides an interface that the Physical layer must abide by. The Data Link Physical Bridge must be implemented by the Physical layer. This implementation must follow the Physical Bridge's specifications.
At the Physical Bridge and Physical layer, the message being received and transmitted is known as a **data stream**. The data stream contains physical header and status information necessary for the protocol to operate correctly.
The following table describes the overhead created by encapsulating a message into a data stream:

Data Stream Header	Data Stream Footer (receive only)[1]
▪ Length field – 1 byte	▪ Received Signal Strength Indicator – 1 byte ▪ Cyclic Redundancy Check OK – 1 bit ▪ Link-Quality Indicator – 7 bits

1 bytes	2 bytes

[1] These bytes are not actually transmitted. When a complete data stream is received, they are appended to the hardware RX FIFO.

PHYSICAL ADDRESSING

Physical addressing provides information necessary to create a connection (link) between two nodes. An address can represent the source or destination address. Creating a link between two nodes is also commonly known as *pairing*.

When the network is being brought up, each node will need to connect to the available nodes around it. To do this, a pairing sequence must be performed.

Simplex communication is one-way from End Point to Gateway through a broadcast. The Gateway can choose to ignore these messages at any time. For simplex communication, the End Point node may use the broadcast address (0) or a fixed address to communicate with the Gateway node.

A pairing request begins by an End Point broadcasting a *request*. The Gateway is always listening for incoming messages. Since the request is done using a broadcast address, the Gateway, if within range, will hear the message.

Once the Gateway receives a request to pair, it will perform the necessary actions to generate the appropriate response. A Gateway may either *accept* the request or *deny* it. In either circumstance, the Gateway will send a response to the requesting End Point node.

MEDIA ACCESS CONTROL (MAC)

At the Data Link layer, messages being received and transmitted are known as **frames**. A frame encapsulates the application packet into a message that the receiving Data Link layer can understand. The frame is made up of header and footer information. The header and footer are wrapped around the application packet.

The following table describes the overhead created by encapsulating a message into a frame:

Frame Header	**Frame Footer (not currently supported)[1]**
▪ PAN identifier – n bytes ▪ Destination address – n bytes ▪ Source address – n bytes ▪ Control – 1 byte ▪ Sequence number – 1 byte	▪ Frame check sequence – 2 bytes
3n+2 bytes	2 bytes

[1] The Frame Check Sequence (FCS) is not currently supported.

Construction consists of encapsulating the payload into a frame. Header information is setup based on the frame type being constructed and its requirements.

When a frame is ready to be sent, the overall size must be calculated. The size of the frame is dependent on the payload being transmitted. The total frame size is the sum of the frame header, payload, and frame footer in bytes.

Once the size of the frame is known and the frame is provided, the Data Link layer can send the frame to the Physical layer. The Physical layer will then turn the frame into a data stream and transmit it over-the-air.
Incoming frames can be received only when the node is listening. Listening is not a blocking operation. Once listening begins, the application can continue operating as normal. A message will be generated and a callback function will be invoked if and when a frame is received. This message will indicate success (if frame received), or failure (if timeout occurs).
A node may filter on an address to be notified only when frames from that address are received.

1.5. REGULATORY COMPLIANCE

The device/end product on which this software driver is used is subject to meeting regulatory compliance (FCC, IC, ETSI, etc.) based on the selected frequency band and the deployed region. Compliance depends on many different factors and is outlined for each of Anaren's AIR proprietary modules. Using this library unaltered and as intended based on library documentation, examples etc. allows the use of the Anaren FCC/IC ID as described in the applicable modules user's guide, for the "intentional radiator" part of certification. For Europe (ETSI) additional rules related to duty cycling or AFA+LBT must be guaranteed and documented by the integrator/OEM. Please refer to the applicable module's user's guide available at: http://www.anaren.com/air/products/ . Using this library without an Anaren AIR module is permitted, but regulatory compliance is not suggested or guaranteed even if applicable Anaren AIR module guidelines are used and the Anaren FCC/IC and ETSI IDs and reports may not be used with such product.

1.6. SOFTWARE LICENSING AGREEMENT

The software is provided as-is under the **GNU Lesser General Public License, version 3.0 (LGPL-3.0)**. This license allows users to use the software under certain restrictions. For more information on the licensing agreement, please visit http://www.gnu.org/copyleft/lesser.html.
Please contact air@anaren.com with any comments or questions for this document.

2. HARDWARE INTERFACE

2.1. HARDWARE ABSTRACTION LAYER

The Hardware Abstraction Layer (HAL) provides an interface for porting between different hardware platforms. This interface defines all of the required functionality that the protocol requires from the underlying hardware; it acts as a contract between the protocol and hardware. The HAL interface is referred to as the Hardware Platform Physical Bridge.
Different Hardware Platform Physical Bridge interfaces exist based on the underlying radio hardware. For instance, the A110x2500PhyBridge exists to work with A110x2500 modules (e.g. A1101R09). These modules require a SPI interface and a General Digital Output (GDO) for interrupts. Other radios may require different forms of interfacing, such as a System-on-Chip (SoC) with an internal set of registers for

the radio hardware. For more information regarding the A110x2500PhyBridge please refer to the CC110x2500 Device Driver Programmer's Guide.
The protocol requires a tick (refer to Introduction – Physical Layer for more information). This tick is used for various operations. To perform these operations, a timer hardware peripheral must be dedicated to the protocol. The following interface must be implemented in the hardware abstraction layer:

- **Hardware timer initialization** – setup timer to tick rate and perform any other necessary operations to initialize it.
- **Hardware timer start** – start the hardware timer count.
- **Hardware timer stop** – stop the hardware timer count (mainly used for power management).

The following provides an illustration of the A110x2500 HAL interface. This interface defines the functions that must be implemented by hardware peripherals and drivers for the protocol to function on the platform. Actual implementation details have been excluded. Please refer to Appendix B and the A110x2500 Module Driver Programmer's Guide for more information on how to port the A110x2500 Physical Bridge to a different platform.

```
// A110x2500 RF serial peripheral interface (SPI)

void A110x2500SpiInit();
void A110x2500SpiRead(unsigned char address, unsigned char *buffer, unsigned char count);
void A110x2500SpiWrite(unsigned char address, const unsigned char *buffer, unsigned char count);

// A110x2500 RF general digital output (GDO)

void A110x2500Gdo0Init();
bool A110x2500Gdo0Event(unsigned char event);
void A110x2500Gdo0WaitForAssert();
void A110x2500Gdo0WaitForDeassert();
enum eCC1101GdoState A110x2500Gdo0GetState();
void A110x2500Gdo0Enable(bool en);

// Hardware Timer

void A110x2500HwTimerInit();
void A110x2500HwTimerStart();
void A110x2500HwTimerStop();
```

3. CONFIGURATION

A pre-included configuration is required to setup and operate the protocol. This configuration file is specific to the application and node type (e.g. End Point, Gateway, etc.). The categories of configuration include: microcontroller global interrupt control support, platform characteristics, radio characteristics, and protocol characteristics.

3.1. MICROCONTROLLER GLOBAL INTERRUPT CONTROL

The protocol performs most of its operations inside of an interrupt service routine (specifically the radio hardware's ISR). This is due to the fact that the protocol must be able to react quickly to events associated with the radio. In order to guarantee that it doesn't block other interrupt-driven operations (e.g. UART), the protocol may enable all other interrupts inside of the radio ISR. To do this, the protocol defines an interface that it uses to enable and disable global interrupts.

The following macros must be defined inside of the configuration file for microcontroller global interrupt control support to exist:

`MCU_DISABLE_INTERRUPT()` – disables global interrupts.
`MCU_ENABLE_INTERRUPT()` – enables global interrupts.
`MCU_CRITICAL_SECTION(code)` – declares a critical section with `code` being the code executed inside the critical section. A typical implementation involves:

1. Get the current interrupt state and store it in a variable.
2. Disable global interrupts.
3. Perform the `code` that must be in the critical section (atomic operation).
4. Return the interrupt state to its original state that's stored in the variable from step 1.

3.2. PLATFORM CHARACTERISTICS

Platform characteristics are for definitions that are specific to the hardware port. For instance, a port exists for Anaren's A110x2500 Booster Pack platform with a specific microcontroller. Anaren's A110x2500 Booster Pack platform allows for traces to be cut for re-routing of CSn and GDO0 signals (please refer to the A110LR09x AIR Booster Pack User's Manual). The port file, which is a C file that implements the A110x2500 Physical Bridge interface, has definitions for automatically re-routing these signals to one of the other positions; no additional changes need to be made other than defining which position the application requires.
The following is an excerpt from a configuration file that is choosing some available positions for CSn and GDO0 as well as choosing a hardware timer for the protocol.

```
#define RF_SPI_CSN_1                    // Default CSn position
#define RF_GDO0_1                       // Default GDO0 position
#define TIMER0_A                        // Protocol uses Timer0_A3
```

Note It is important to remember that these definitions only exist if the port file, which must be created for each platform, has the option available. From the excerpt above, the port file has already defined and provided implementations for the different hardware timers (please refer to the Hardware Abstraction Layer for more information). If this work had not been done in the port file, the implementation in the port file would need to be changed if the default timer wasn't the desired selection (e.g. `Timer1_A` was selected as the protocol timer but the port only provides `Timer0_A` implementation.

Platform characteristics may not exist in the port file. Please refer to the selected platform port file (if a custom port was not created).

3.3. RADIO CHARACTERISTICS

An Anaren AIR module must be defined. This module selection dictates the configurations and output power that are available to the module (defined in the module driver). For instance, if the application is using the A110x2500 Physical Bridge, the specific module must be chosen (e.g. `A1101R09_MODULE`). Possible modules are referenced below:

A110x2500 Physical Bridge:

- `A1101R04_MODULE`
- `A1101R08_MODULE`
- `A1101R09_MODULE`
- `A110LR09_MODULE`
- `A2500R24_MODULE`

Based on the radio module selected, a set of configuration values must be defined (please refer to Platform Characteristics for more information on selecting a module for the appropriate hardware platform physical bridge). When using Anaren AIR modules, the possible settings are defined in the appropriate programmer's guide (e.g. A110x2500 Module Driver Programmer's Guide).
The required configuration information includes the default configuration (e.g. radio module certified register settings from Anaren's AIR module driver), output power setting, and the physical hardware's maximum transmit FIFO size (in bytes).

3.4. PROTOCOL CHARACTERISTICS

The protocol characteristics define how the protocol should operate. This includes the node role, size of addressing (including PAN identifier), and the maximum payload length.
Protocol node types include: `PROTOCOL_GATEWAY` and `PROTOCOL_ENDPOINT`. An End Point node can be classified as either a Simplex (one-way) node or a Half Duplex (two-way) node. In order for an End Point node to utilize two-way features in the protocol, PROTOCOL_USE_SYNC_TIMEOUT must be defined. This definition is used to include power saving timeouts that are used when the End Point node goes into a receive state expecting a response from the Gateway node. It is an absolute requirement for half duplex End Point nodes.
For addressing, the user may specify the length (in bytes) of the PAN identifier and the address of the node. The `PROTOCOL_PHYADDRESS_PANID_SIZE` defines the size of the Personal Area Network (PAN) identifier in bytes, and `PROTOCOL_PHYADDRESS_ADDRESS_SIZE` defines the size of the local address.

Note The PAN identifier size must match for all nodes in the network. Likewise, the address size for all nodes must also be identical. The frame overhead changes based on these factors. If nodes use different sizes, there is no guarantee that communication will be considered valid.

The frame payload length is dictated by `PROTOCOL_FRAME_MAX_PAYLOAD_LENGTH`. This value is used as a method for declaring the maximum size a frame can ever be. The protocol creates a frame structure based on this payload size for sending and receiving frames. The total size of a frame is the frame overhead size and the frame payload size.

Note The frame total size should not be larger than the physical hardware RX/TX FIFOs.

4. BASICS TUTORIAL

This section demonstrates basic usage of the protocol. It provides code snippets that can be used to **initialize** the protocol, perform the role of a **Gateway** node, perform the role of a **Simplex End Point** node, and perform the role of a **Half Duplex End Point** node.
For the examples listed below, it is assumed that the platform implementation of the hardware abstraction layer is defined. It is also assumed that a configuration file is defined and pre-included for the current application. Other assumptions are defined in the corresponding tutorial.

4.1. INITIAL SETUP

To setup the protocol, an information structure must be created and passed to the initialization function. The initialization function handles setting up all required protocol structures and hardware through the hardware platform physical bridge. The following snippet of code illustrates setting up a node without any callback functions implemented:

Note It is assumed that the configuration file has defined the PAN identifier and address size to be 1 byte for each.

```
static const struct sProtocolSetupInfo gProtocolSetupInfo = {
  { 0x01 },  // Physical address PAN identifier
  { 0x01 },  // Physical address
  NULL,      // Protocol Backup callback (not used)
  NULL       // Protocol Data Transfer Complete callback (not used)
};

...

int main(void)
{
  // Setup the hardware platform.
  // Disable global interrupts until the protocol has been initialized.
  ...
  ProtocolInit(&gProtocolSetupInfo);
  ...
  // Re-enable global interrupts once the protocol and hardware is initialized.
  ...
  return 0;
}
```

4.2. GATEWAY NODE

A Gateway node is the “sink” for most information in the network. The following snippet illustrates the essential features required by a Gateway node that may receive simplex and half duplex data from End Point nodes.

```
// Callback function prototype
unsigned char TransferComplete(bool dataRequest, unsigned char *data, unsigned char length);

...

static const struct sProtocolSetupInfo gProtocolSetupInfo = {
  { 0x01 },        // Physical address PAN identifier
  { 0x01 },        // Physical address
  NULL,            // Protocol Backup callback (not used)
  TransferComplete // Protocol Data Transfer Complete callback (not used)
};
```

```
...
unsigned char TransferComplete(bool dataRequest, unsigned char *data, unsigned char length)
{
  ...
  /**
   *  Read incoming data that exists in *data. Use length to determine the number of bytes.
   *  The *data refers to an internal buffer. If this data is not retrieved, it will be lost
   *  upon returning from this callback. Optionally, check status information, such as source
   *  address (please refer to API).
   */
  ...
  if (dataRequest)
  {
    ...
    // If the node is requesting data and data is available, load it to be sent.
    ProtocolLoadDataRequest(...);
    ...
  }
  ...
}

int main(void)
{
  // Setup the hardware platform.
  // Disable global interrupts until the protocol has been initialized.
  ...
  ProtocolInit(&gProtocolSetupInfo);
  ...
  // Re-enable global interrupts once the protocol and hardware is initialized.
  ...
  return 0;
}

// Protocol Engine Interrupt Service Routine
{
  ...
  ProtocolEngine(...);
  ...
}
```

4.3. SIMPLEX END POINT NODE

As an End Point node, the choice can be made for which type of transfers the node will perform. Simplex transfers are the simplest and fastest methods for sending information. No connection is even required with the Gateway node (e.g. these transfers are broadcasts, but may be directed to a certain PAN identifier). The following snippet illustrates using the `ProtocolSimpleTransfer` API call to perform simplex data transfers.

```
...
while (true)
{
  ...
  if (!ProtocolSimpleTransfer(...))
  {
    // Loop, go to sleep, or do something else until the transfer is complete.
  }
  ...
}
...
```

4.4. HALF DUPLEX END POINT NODE

Half duplex data transfers consist of the End Point node sending a message to the Gateway node and then going into a receive state to wait for a reply before ending the communication. Two-way communication requires that the End Point node connect to the Gateway node and acquire the Gateway node's address. To do this, a connect API call must be performed prior to performing an transfers. This is illustrated in the code snippet below:

Note It is assumed that the configuration file has been set appropriately to accommodate for a half-duplex End Point node.

```
...
while (!ProtocolStatusPhyAddressInfo().connected)
{
  if (!ProtocolConnect(...))
  {
    // Loop, go to sleep, or do something else until the connection is established.
  }
}

while (true)
{
  ...
  if (!ProtocolTransfer(...))
  {
    // Loop, go to sleep, or do something else until the transfer is complete.
  }
  ...
}
...
```

5. APPLICATION PROGRAMMING INTERFACE REFERENCE

This section describes the Application Programming Interface (API) structures and functions.

5.1. STRUCTURES

5.1.1. SETUP INFORMATION

The setup information structure is used to initialize the protocol with specific parameters. These parameters include addressing information and callback function pointers. The callback functions may have different requirements based on whether the node is considered a Gateway node or an End Point node.

Note This structure may be stored in RAM or ROM depending on the application requirements.

The `Backup` callback function should be used as a mechanism for storing local node addressing information in a non-volatile backup. This backup is to be used as a way of recovering the network if power were to be completely lost on a node. Boot up uses this `Backup` as a way of retrieving lost information (in RAM) from a non-volatile source such as an EEPROM or flash device. Data for the backup mechanism should be formatted as follows:

Fixed address flag	Pan id	Local address	Remote address
1 byte	n-bytes	m-bytes	m-bytes

Note The `Backup` callback is only available for End Point nodes. It is only intended to be used with nodes performing half-duplex transfers (that require a connect handshake).

The `LinkRequest` callback function should be used as a notification of a link request event. This allows the application to read optional request payload information and accept/deny the request.

Note The `LinkRequest` callback is only available for Gateway nodes.

The `TransferComplete` callback function should be used as a notification of a transfer complete event. This allows the application to read an incoming message and act on it.

Note The `TransferComplete` callback on a Gateway node has the ability to send data back if the End Point node has requested data (using the `dataRequest` flag indicator). This data response is optional and application-dependent.

```
#if defined( PROTOCOL_ENDPOINT )
struct sProtocolSetupInfo
{
  // Physical addressing setup
  unsigned char panId[PROTOCOL_PHYADDRESS_PANID_SIZE];
  unsigned char address[PROTOCOL_PHYADDRESS_ADDRESS_SIZE];
  bool(*Backup)(bool read, unsigned char *data, unsigned char size);
  unsigned char(*TransferComplete)(unsigned char *payload, unsigned char length);
};
#elif defined( PROTOCOL_GATEWAY )
struct sProtocolSetupInfo
{
  unsigned char panId[PROTOCOL_PHYADDRESS_PANID_SIZE];
  unsigned char address[PROTOCOL_PHYADDRESS_ADDRESS_SIZE];
  bool(*LinkRequest)(unsigned char *payload, unsigned char length);
  unsigned char(*TransferComplete)(bool dataRequest,
                                   unsigned char *payload,
                                   unsigned char length);
};
#endif
```

5.1.2. STATUS INFORMATION

Status information is provided to the application for debugging and general use. It features Data Link and Physical layer status information.

```
struct sProtocolStatusInfo
{
  struct sProtocolDataLinkInfo
  {
    struct sProtocolPhyAddressInfo
    {
      unsigned char panId[PROTOCOL_PHYADDRESS_PANID_SIZE];  // Personal Area Network (PAN) ID
      #if defined( PROTOCOL_ENDPOINT )
      bool connected;                                    // Connection status
      #endif
    } phyAddressInfo;
    struct sProtocolFrameInfo
    {
      unsigned char srcAddr[PROTOCOL_PHYADDRESS_ADDRESS_SIZE];  // Source of the payload
      unsigned char seqNumber;                                  // Frame sequence number
    } frameInfo;
  } dataLink;

  struct sProtocolPhysicalInfo
  {
    struct sProtocolDataStreamInfo
    {
      signed char rssi;         // Received signal strength indicator
      unsigned char status;     // Status [CRC(1):LQI(7)]
    } dataStreamInfo;
  } physical;
};
```

5.2. FUNCTIONS

5.2.1. CONFIGURATION AND STATUS

ProtocolInit(setup)
Initializes the protocol structures and available hardware (e.g. communication and timer peripherals).

Parameters:	**setup:** const struct sProtocolSetupInfo*
	Setup information required to properly initialize the protocol.
Returns:	**bool**
	Success of initializing the protocol. If false, setup structure was NULL.

Examples:
```
// This setup is for an End Point node.
const struct sProtocolSetupInfo gProtocolSetupInfo = {
  { PROTOCOL_CHANNEL_LIST },      // Physical channel list
  { PROTOCOL_PHYSICAL_PAN_ID }, // Physical address PAN identifier
  { PROTOCOL_PHYSICAL_ADDRESS },// Physical address
  NULL,                         // Protocol Backup callback (not used)
  NULL                          // Protocol Data Transfer Complete callback (not used)
};
...
int main(void)
{
  ...
  // Attempt to initialize protocol hardware and information using the provided
  // setup structure data.
  if (!ProtocolInit(&gProtocolSetupInfo))
  {
    // Fatal Error: Failed to initialize the protocol.
  }
  ...
}
```

ProtocolStatusPhyAddressInfo()
Retrieves information regarding the protocol data link layer physical address.
Parameters: None

Returns: **struct sProtocolPhyAddressInfo**
PhyAddress information structure with information regarding the protocol's physical addressing information. For more information, please refer to Status Information.

Examples:
```
while (!ProtocolStatusPhyAddressInfo().connected)
{
  if (!ProtocolConnect(NULL, 0))
  {
    // Put the microcontroller into a low power state (sleep).
  }
}
```

ProtocolStatusFrameInfo()
Retrieves information regarding the protocol data link layer frame.
Parameters: None
Returns: **struct sProtocolFrameInfo**
Frame information structure with information regarding the protocol's frame information. For more information, please refer to Status Information.
Examples: None

ProtocolStatusPhysicalInfo()
Retrieves information regarding the protocol physical layer.
Parameters: None
Returns: **const struct sProtocolPhysicalInfo***
Physical information structure with information regarding the protocol's physical layer information. For more information, please refer to Status Information.
Examples: None

ProtocolBusy()
Flag indicator that can be used to determine if the protocol is currently busy performing an operation.
Parameters: None
Returns: **bool**
Indicates if the protocol is currently busy (true) or ready (false) for the next operation.
Examples:
```
if (!ProtocolBusy())
{
  // Perform an operation that can only be done if the protocol is not busy.
}
```

5.2.2. CONNECTION MANAGEMENT

ProtocolConnect(txData, length)

Attempts to connect the local node to a remote node. If a connection does not exist, send a request. This process is asynchronous and will not block. Polling must be performed to check if a connection exists.

Note This function is only supported by End Point nodes.

Parameters: **txData:** const unsigned char*
Data buffer to be transferred during the connection attempt. This message is intended to be used for passing connection specific information to the remote node (e.g. a light switch establishing a connection with a light fixture).
length: unsigned char
Number of bytes in the data buffer.

Returns: **bool**
Success of the operation. If a connection has been made, return true. Otherwise, return false if a connection does not exist.

Examples:

```
while (!ProtocolStatusPhyAddressInfo().connected)
{
  if (!ProtocolConnect(NULL, 0))
  {
    // Put the microcontroller into a low power state (sleep).
  }
}
```

ProtocolDisconnect()
Disconnects the local node from a remote node (if a connection exists).

Note This function is only supported by End Point nodes.

Parameters: None
Returns: Nothing
Examples: None

5.2.3. DATA TRANSFERS

ProtocolSimpleTransfer(txData, length)
Initiates a simple (simplex) protocol data transfer. This function can be used as a “brute force” mechanism for transferring a message as no connection is required (please refer to Connection Management for more information on connections). This process is asynchronous and will not block.

Note This function is only supported by End Point nodes.

Parameters: **txData:** const unsigned char*
Data buffer to be transferred.
length: unsigned char

Number of bytes in the data buffer.

Returns: **bool**

Success of the operation. If the protocol is not busy, return true. Otherwise, the protocol is busy, return false.

Examples:

```
struct sPacket
{
  unsigned char payload[10]; // Packet payload
};
...
const struct sPacket gPacket = {
  "Hello"                    // Set the initial payload to a "Hello" string
};
...
// Perform a simple transfer of the packet.
if (!ProtocolSimpleTransfer((unsigned char*)&gPacket, sizeof(struct sPacket)))
{
  // Put the microcontroller into a low power state (sleep).
}
```

ProtocolTransfer(txData, length)

Initiates a standard (half-duplex) protocol data transfer. This process is asynchronous and will not block.

Note	This function is only supported by End Point nodes.

Note	If an End Point is not connected to a Gateway node the provided data will not be transferred until a connection exists. Be sure to ALWAYS call `ProtocolConnect` before attempting to use this function.

Parameters: **txData:** const unsigned char*

Data buffer to be transferred.

length: unsigned char

Number of bytes in the data buffer.

Returns: **bool**

Success of the operation. If the protocol is not busy, return true. Otherwise, the protocol is busy or no call to `ProtocolConnect` was made, return false.

Examples:

```
while (!ProtocolStatusPhyAddressInfo().connected)
{
  if (!ProtocolConnect(NULL, 0))
  {
    // Put the microcontroller into a low power state (sleep).
  }
}
...
// Perform a data transfer of the packet.
if (!ProtocolTransfer((unsigned char*)&gPacket, sizeof(struct sPacket)))
{
  // Put the microcontroller into a low power state (sleep).
}
```

ProtocolLoadDataResponse(txData, length)

Loads a response to a data request into the protocol transmission buffer.

Note	This function is only supported by Gateway nodes.

Note	This function should be used in conjunction with data requests only. When a remote End Point node requests data, load any new available data using this function.

Note	This function must be called from inside the `TransferComplete` callback function. For more information on the `TransferComplete` callback, please refer to Status Information.

Parameters: **txData:** const unsigned char*
Data buffer to be transferred in a response message.
length: unsigned char
Number of bytes in the data buffer.

Returns: Nothing

Examples:

```
struct sPacket
{
  unsigned char payload[10]; // Packet payload
};
...
static struct sPacket gPacketTx = {
  "Hello"                   // Set the initial payload to a "Hello" string
};
static struct sPacket gPacketRx = {
  ""                     // Set the initial payload to an empty string
};
...

unsigned char TransferComplete(bool dataRequest,
                               unsigned char *data,
                               unsigned char length)
{
  // Cast the received data pointer to a packet structure pointer so that it may
  // be accessed using the structure member notation.
  struct sPacket *p = (struct sPacket*)data;

  // Retrieve the sequence number and copy the payload from the protocol into
  // the local application packet (gPacket).
  memcpy(gPacketRx.payload, p->payload, length);

  // Check if a data request has been made.
  if (dataRequest)
  {
    // Provide any available data in a "data request" response transfer. It is assumed that all
    // data processing has been performed and is available to send.
    ProtocolLoadDataResponse((unsigned char*)&gPacketTx, sizeof(struct sPacket));
  }

  return 0;
}
```

5.2.4. INTERRUPT SERVICE ROUTINES

ProtocolEngine(event)

Performs main operations for the protocol.

Note It is assumed that this function is called inside an interrupt service routine where global interrupts are DISABLED. The protocol engine will re-enable global interrupts when it is ready to do so during this operation.

Parameters: **event:** volatile unsigned char
Protocol action flag indicator. In most instances, this will be the protocol's physical hardware's port interrupt flag. It is used to determine if a protocol event has taken place and needs to be serviced in a I/O interrupt service routine.

Returns: **unsigned char**
Reserved use.

Examples:

```
{
  // I/O interrupt service routine.
  ...
  // Store the port interrupt flag register in an event variable.
  ...
  /**
   *  Notify the protocol of a port event. The protocol will determine if it is
   *  associated with the GDO0 pin and act accordingly (e.g. if GDO0 has not
   *  triggered an interrupt then the protocol will not run, otherwise it will
   *  continue where it left off).
   *
   *  Note: Clearing of the GDO0 event (interrupt flag bit) is handled
   *  internally. It is important that the application does not clear the GDO0
   *  event in this ISR.
   */
  ProtocolEngine(event);
  ...
}
```

ProtocolEngineTick()

Performs main operations with the protocol system tick (timer).

Note The Physical Bridge requires a 16-bit hardware timer configured with a tick rate of at least 1ms. The tick rate should be calculated taking the crystal/oscillator error into account.

Parameters: None

Returns: Nothing

Examples:

```
{
  // Hardware timer interrupt service routine (1ms tick).
  ...
  // Notify the protocol of a timer interrupt. The protocol uses this interrupt
  // as a tick to increment any counters that are actively running.
  ProtocolEngineTick();
  ...
}
```

APPENDIX A. EXAMPLE USAGE

SIMPLEX COMMUNICATION

The following example demonstrates simplex (one-way) communication from End Point node(s) to a central Gateway node.

Supported Microcontroller(s):	MSP430G2553
Supported Platform(s):	A110x2500 Booster Pack
File Dependency:	API.h

The following table outlines memory usage for this example:

Node Type	CODE memory (ROM)	DATA memory (RAM)	CONST memory (ROM)
End Point	4,066 bytes	167 bytes	110 bytes
Gateway	3,904 bytes	169 bytes	100 bytes

Note Compiled code with IAR C/C++ Compiler for MSP430 5.50.1 (5.50.1.50465) and no optimization settings.

END POINT NODE(S)

```
#ifndef bool
#define bool unsigned char
#define true 1
#define false 0
#endif

#include "API.h"

// ----------------------------------------------------------------------------
/**
 *  Defines, enumerations, and structure definitions
 */

#define ST(X) do { X } while (0)

/**
 *  Abstract hardware based on the supported example platform being used.
 *
 *  Currently supported platforms:
 *    - TI MSP430G2553 + AIR A110x2500 Booster Pack (BPEXP430G2x53.c)
 */
#if defined( __MSP430G2553__ )
#define HardwareInit()\
  ST\
  (\
    WDTCTL = WDTPW | WDTHOLD;\
    BCSCTL1 = CALBC1_8MHZ;\
    DCOCTL = CALDCO_8MHZ;\
  )
#define McuSleep()     _BIS_SR(LPM4_bits | GIE)   // Go to low power mode 4
#define McuWakeup()    _BIC_SR(LPM4_EXIT);        // Wake up from low power mode 4
#define GDO0_VECTOR    PORT2_VECTOR
#define GDO0_EVENT     P2IFG
#endif

/**
 *  sPacket - an example packet. The sequence number is used to demonstrate
 *  communication by sending the same message (payload) and incrementing the
 *  sequence number on each transmission.
 */
struct sPacket
{
  unsigned char seqNum;     // Packet sequence number
  unsigned char payload[9]; // Packet payload
};

// ----------------------------------------------------------------------------
/**
 *  Global data
 */

/**
 *  The following instance of sProtocolSetupInfo is used to initialize the
 *  protocol with required parameters. The End Point node role has the following
 *  parameters [optional parameters are marked with an asterix (*)].
 *
 *  { channel_list, pan_id, local_address, Backup(*), TransferComplete(*) }
 *
 *  Note: Parameters marked with the asterix (*) may be assigned "NULL" if they
 *  are not needed.
 */
static const struct sProtocolSetupInfo gProtocolSetupInfo = {
  { PROTOCOL_CHANNEL_LIST },// Physical channel list
  { 0x01 },                 // Physical address PAN identifier
  { 0x02 },                 // Physical address
  NULL,                     // Protocol Backup callback (not used)
  NULL                      // Protocol Data Transfer Complete callback (not used)
};

static struct sPacket gPacket = {
  0x00,                     // Set the initial sequence number value to 0
  "Hello"                   // Set the initial payload to a "Hello" string
};

// ----------------------------------------------------------------------------

/**
 *  PlatformInit - sets up platform and protocol hardware. Also configures the
 *  protocol using the setup structure data.
 *
 *    @return   Success of the operation. If true, the protocol has been setup
 *              successfully. If false, an error has occurred during protocol
 *              setup.
 */
bool PlatformInit(void)
{
  // Disable global interrupts during hardware initialization to prevent any
  // unwanted interrupts from occurring.
  MCU_DISABLE_INTERRUPT();

  // Setup basic platform hardware (e.g. watchdog, clocks).
  HardwareInit();

  // Attempt to initialize protocol hardware and information using the provided
  // setup structure data.
  if (!ProtocolInit(&gProtocolSetupInfo))
  {
    return false;
  }

  // Re-enable global interrupts for normal operation.
  MCU_ENABLE_INTERRUPT();
  return true;
}

/**
 *  main - main application loop. Sets up platform and then performs simple
 *  transfers (simplex) while incrementing the sequence number for the lifetime
 *  of execution.
 *
 *    @return   Exit code of the main application, however, this application
 *              should never exit.
 */
int main(void)
{
  // Setup hardware and protocol.
  PlatformInit();

  while (true)
  {
    // Perform a simple transfer of the packet.
    if (!ProtocolSimpleTransfer((unsigned char*)&gPacket, sizeof(struct sPacket)))
    {
      // Put the microcontroller into a low power state (sleep). Remain here
      // until the ISR wakes up the processor.
      McuSleep();
    }

    /**
     *  Check if the protocol is busy. If it is, a new transfer cannot occur
     *  until it becomes ready for the next instruction. Do not increment the
     *  sequence number until the protocol is ready. This prevents incrementing
     *  the sequence number more than once between transmissions.
     */
    if (!ProtocolBusy())
    {
      // Increment the sequence number for the next transmission.
      gPacket.seqNum++;
    }
  }
}

/**
 *  GDO0Isr - GDO0 interrupt service routine. This service routine will always
 *  be a I/O interrupt service routine. Therefore, it is important to pass the
 *  port flag to the ProtocolEngine so that the protocol can determine if a GDO0
 *  event has occurred.
 */
#pragma vector=GDO0_VECTOR
__interrupt void GDO0Isr(void)
{
  /**
   *  Store the port interrupt flag register so that it may be used to determine
   *  what caused the interrupt (e.g. did a GDO0 interrupt occur? The protocol
   *  needs this information to determine what to do next...).
   */
  register volatile unsigned char event = GDO0_EVENT;

  /**
   *  Notify the protocol of a port event. The protocol will determine if it is
   *  associated with the GDO0 pin and act accordingly (e.g. if GDO0 has not
   *  triggered an interrupt then the protocol will not run, otherwise it will
   *  continue where it left off).
   *
   *  Note: Clearing of the GDO0 event (interrupt flag bit) is handled
```

```
     *  internally. It is important that the application does not clear the GDO0
     *  event in this ISR.
     */
    ProtocolEngine(event);

    // Wake up the microcontroller to continue normal operation upon exiting the
    // ISR.
    McuWakeup();
  }

  // Note: No hardware timer interrupt required for this example because the
  // End Point node does not perform half duplex transfers.
```

GATEWAY NODE

```
#ifndef bool
#define bool unsigned char
#define true 1
#define false 0
#endif

#include <string.h>       // memcpy
#include "API.h"

// ----------------------------------------------------------------------------
/**
 *  Defines, enumerations, and structure definitions
 */

#define ST(X) do { X } while (0)

/**
 *  Abstract hardware based on the supported example platform being used.
 *
 *  Currently supported platforms:
 *    - TI MSP430G2553 + AIR A110x2500 Booster Pack (BPEXP430G2x53.c)
 */
#if defined( __MSP430G2553__ )
#define HardwareInit()\
  ST\
  (\
    WDTCTL = WDTPW | WDTHOLD;\
    BCSCTL1 = CALBC1_8MHZ;\
    DCOCTL = CALDCO_8MHZ;\
  )
#define McuSleep()     _BIS_SR(LPM4_bits | GIE)  // Low power mode 4
#define GDO0_VECTOR    PORT2_VECTOR
#define GDO0_EVENT     P2IFG
#endif

/**
 *  sPacket - an example packet. The sequence number is used to demonstrate
 *  communication by sending the same message (payload) and incrementing the
 *  sequence number on each transmission.
 */
struct sPacket
{
  unsigned char seqNum;     // Packet sequence number
  unsigned char payload[9]; // Packet payload
```

```
};

// ----------------------------------------------------------------------------
/**
 *  Callback function prototypes
 */

/**
 *  TransferComplete - acts as the callback function for the Protocol Data
 *  Transfer Complete event. When a frame is received that meets addressing
 *  requirements, the Protocol Data Transfer Complete event is triggered and a
 *  callback must be implemented to receive the frame payload (packet).
 *
 *  Note: Please refer to API.h for more information on the TransferComplete
 *  callback for a Gateway node.
 */
unsigned char TransferComplete(bool dataRequest,
                               unsigned char *data,
                               unsigned char length);

// ----------------------------------------------------------------------------
/**
 *  Global data
 */

/**
 *  The following instance of sProtocolSetupInfo is used to initialize the
 *  protocol with required parameters. The Gateway node role has the following
 *  parameters [optional parameters are marked with an asterix (*)].
 *
 *  { channel_list, pan_id, local_address, LinkRequest(*), TransferComplete(*) }
 *
 *  Note: Parameters marked with the asterix (*) may be assigned "NULL" if they
 *  are not needed.
 */
static const struct sProtocolSetupInfo gProtocolSetupInfo = {
  { PROTOCOL_CHANNEL_LIST },// Physical channel list
  { 0x01 },                 // Physical address PAN identifier
  { 0x01 },                 // Physical address
  NULL,                     // Protocol Link Request callback (not used)
  TransferComplete          // Protocol Data Transfer Complete callback
};

static struct sPacket gPacket = {
  0x00,                     // Set the initial sequence number value to 0
  ""                        // Set the initial payload to an empty string
};

// ----------------------------------------------------------------------------

unsigned char TransferComplete(bool dataRequest,
                               unsigned char *data,
                               unsigned char length)
{
  // Cast the received data pointer to a packet structure pointer so that it may
  // be accessed using the structure member notation.
  struct sPacket *p = (struct sPacket*)data;

  // Retrieve the sequence number and copy the payload from the protocol into
  // the local application packet (gPacket).
  gPacket.seqNum = p->seqNum;
  memcpy(gPacket.payload, p->payload, length-1);

  return 0;
```

```
}

/**
 *  PlatformInit - sets up platform and protocol hardware. Also configures the
 *  protocol using the setup structure data.
 *
 *    @return   Success of the operation. If true, the protocol has been setup
 *              successfully. If false, an error has occurred during protocol
 *              setup.
 */
bool PlatformInit(void)
{
  // Disable global interrupts during hardware initialization to prevent any
  // unwanted interrupts from occurring.
  MCU_DISABLE_INTERRUPT();

  // Setup basic platform hardware (e.g. watchdog, clocks).
  HardwareInit();

  // Attempt to initialize protocol hardware and information using the provided
  // setup structure data.
  if (!ProtocolInit(&gProtocolSetupInfo))
  {
    return false;
  }

  // Re-enable global interrupts for normal operation.
  MCU_ENABLE_INTERRUPT();

  return true;
}

/**
 *  main - main application loop. Sets up platform and then goes to sleep for
 *  the lifetime of execution.
 *
 *    @return   Exit code of the main application, however, this application
 *              should never exit.
 */
int main(void)
{
  // Setup hardware and protocol.
  PlatformInit();

  /**
   *  For this example, all operations are performed inside the protocol ISR
   *  (ProtocolEngine) and callback function (TransferComplete). The protocol
   *  puts the Gateway node into a receive state by default. The application
   *  main loop can simply sleep.
   */
  while (true)
  {
    // Put the microcontroller into a low power state (sleep).
    McuSleep();
  }
}

/**
 *  GDO0Isr - GDO0 interrupt service routine. This service routine will always
 *  be a I/O interrupt service routine. Therefore, it is important to pass the
 *  port flag to the ProtocolEngine so that the protocol can determine if a GDO0
 *  event has occurred.
 */
#pragma vector=GDO0_VECTOR
```

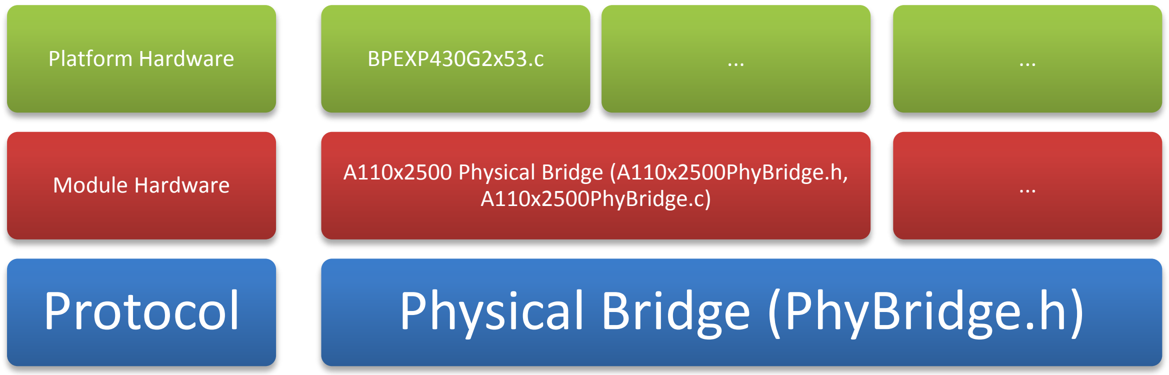
```
__interrupt void GDO0Isr(void)
{
  /**
   *  Store the port interrupt flag register so that it may be used to determine
   *  what caused the interrupt (e.g. did a GDO0 interrupt occur? The protocol
   *  needs this information to determine what to do next...).
   */
  register volatile unsigned char event = GDO0_EVENT;

  /**
   *  Notify the protocol of a port event. The protocol will determine if it is
   *  associated with the GDO0 pin and act accordingly (e.g. if GDO0 has not
   *  triggered an interrupt then the protocol will not run, otherwise it will
   *  continue where it left off).
   *
   *  Note: Clearing of the GDO0 event (interrupt flag bit) is handled
   *  internally. It is important that the application does not clear the GDO0
   *  event in this ISR.
   */
  ProtocolEngine(event);
}

// Note: No hardware timer interrupt required for this example because the
// Gateway node does not use it for anything at this time.
```

APPENDIX B. PLATFORM PORT

Porting the protocol from one platform to another is a simple task. In the following example, the protocol is ported to the TI EXP430G2 Launchpad Development Kit.
The protocol hardware abstraction layer is partitioned in the following manner:

- \code\Physical: defines all physical bridges that implement the protocol's Physical Bridge interface (\code\DataLink\PhyBridge). Currently, the protocol only supports A110x2500-based modules (e.g. A1101R09, A110LR09).
- \code\Physical\A110x2500\PhyBridge: defines the interface for communicating with an A110x2500-based module (A110x2500PhyBridge.h); implements the protocol physical bridge (A110x2500PhyBridge.c) using A110x2500 module specific hardware calls (e.g. A1101R09Init, CC1101Transmit, etc.).
- \platforms\A110x2500\MSP430: defines the A110x2500 physical bridge implementation (based on A110x2500PhyBridge.h).

This structure is illustrated below:


Platform Hardware
BPEXP430G2x53.c
...
...
Module Hardware
A110x2500 Physical Bridge (A110x2500PhyBridge.h, A110x2500PhyBridge.c)
...
Protocol
Physical Bridge (PhyBridge.h)

Supported Microcontroller(s):	MSP430G2553
Supported Platform(s):	A110x2500 Booster Pack
File Dependency:	A110x2500PhyBridge.h, msp430g2553.h

Note	The following implementation is identical to the provided platform port for the Launchpad located in \platforms\A110x2500\MSP430\BPEXP430G2x53.c. However, it has been simplified to focus on the implementation of each function using specific MSP430 header file naming conventions instead of re-definitions.

```
#ifndef bool
#define bool unsigned char
#define true 1
#define false 0
#endif
#include "A110x2500PhyBridge.h"

// Supported microcontrollers
#if defined( __MSP430G2553__ )
#include "msp430g2553.h"
#else
#error "Board Error 0100: Selected microcontroller is not supported"
#endif

// ------------------------------------------------------------------------------
/**
 *  Definitions, enumerations, and structures
 */

// ------------------------------------------------------------------------------
/**
 *  Global data
 */

// ------------------------------------------------------------------------------
/**
 *  Private interface
 */

// ------------------------------------------------------------------------------
/**
 *  Public interface
 */

// ------------------------------------------------------------------------------
// A110x2500 RF serial peripheral interface (SPI)

void A110x2500SpiInit()
{
  // Setup CSn line.
  P2DIR |= BIT7;
  P2OUT |= BIT7;
  P2SEL &= ~BIT7;
  P2SEL2 &= ~BIT7;

  // Setup the USCIB0 peripheral for SPI operation.
  // Note: The current implementation assumes the SMCLK doesn't exceed 10MHz.
  UCB0CTL1 |= UCSWRST;
  UCB0CTL0 |= (UCMODE_0 | UCCKPH | UCMSB | UCMST | UCSYNC);
  UCB0CTL1 |= UCSSEL_2;
```

```
    UCB0BR1 = 0;
    UCB0BR0 = 2;

    // Setup SCLK, MOSI, and MISO lines.
    P1SEL |= BIT5 | BIT6 | BIT7;
    P1SEL2 |= BIT5 | BIT6 | BIT7;

    UCB0CTL1 &= ~UCSWRST;
  }

  void A110x2500SpiRead(unsigned char address, unsigned char *buffer, unsigned char count)
  {
    register volatile unsigned char i;    // Buffer iterator
    /**
     *  Note: The buffer iterator should not be used as an offset. If the iterator
     *  were used, some compilers may issue a warning similar to the following:
     *
     *    "...undefined behavior: the order of volatile accesses is undefined in
     *    this statement".
     *
     *  To prevent this, a new variable is introduced to handle acting as an
     *  offset.
     */
    unsigned char j;                      // Buffer offset

    P2OUT &= ~BIT7;
    // Look for CHIP_RDYn from radio.
    while (P1IN & BIT6);

          // Write the address/command byte.
          IFG2 &= ~UCB0RXIFG;
    UCB0TXBUF = address;
    while (!(IFG2 & UCB0RXIFG));
          IFG2 &= ~UCB0RXIFG;

          // Write dummy byte(s) and read response(s).
    for (i = 0, j = 0; i < count; i++, j++)
    {
                 while (!(IFG2 & UCB0TXIFG));
      UCB0TXBUF = 0xFF;
      while (!(IFG2 & UCB0RXIFG));
      *(buffer+j) = UCB0RXBUF;
    }

          // Wait for operation to complete.
    while(UCB0STAT & UCBUSY);
    P2OUT |= BIT7;
  }

  void A110x2500SpiWrite(unsigned char address, const unsigned char *buffer, unsigned char count)
  {
    register volatile unsigned char i;    // Buffer iterator

    P2OUT &= ~BIT7;
    // Look for CHIP_RDYn from radio.
    while (P1IN & BIT6);

          // Write the address/command byte.
    UCB0TXBUF = address;

          // Write data byte(s).
    for (i = 0; i < count; i++)
    {
      while (!(IFG2 & UCB0TXIFG));
      UCB0TXBUF = *(buffer+i);
    }

          // Wait for operation to complete.
    while(UCB0STAT & UCBUSY);
    P2OUT |= BIT7;
  }

  // ----------------------------------------------------------------------------
  // A110x2500 RF general digital output (GDO)

  void A110x2500Gdo0Init()
  {
    P2DIR &= ~BIT6;
    P2IES &= ~BIT6;
    P2SEL &= ~BIT6;
    P2SEL2 &= ~BIT6;
  }

  bool A110x2500Gdo0Event(unsigned char event)
  {
    if (BIT6 & event)
    {
      // Clear GDO0 event.
      P2IFG &= ~BIT6;
      return true;
    }
    return false;
  }

  void A110x2500Gdo0WaitForAssert()
  {
    P2IES &= ~BIT6;
  }

  void A110x2500Gdo0WaitForDeassert()
  {
    P2IES |= BIT6;
  }

  enum eCC1101GdoState A110x2500Gdo0GetState()
  {
    return (P2IES & BIT6)
      ? eCC1101GdoStateWaitForDeassert
        : eCC1101GdoStateWaitForAssert;
  }

  void A110x2500Gdo0Enable(bool en)
  {
    if (en)
    {
      P2IE |= BIT6;
    }
    else
    {
      P2IE &= ~BIT6;
    }
  }

  // ----------------------------------------------------------------------------
  // Hardware Timer

  void A110x2500HwTimerInit()
  {
    TA0CTL = TASSEL_2 | ID_3 | MC_0;
    TA0CCTL0 |= CCIE;
    TA0CCR0 = 1000;
  }

  void A110x2500HwTimerStart()
  {
    TA0CTL &= ~MC_3;
    TA0CTL |= MC_1;
  }

  void A110x2500HwTimerStop()
  {
    TA0CCTL0 &= ~CCIFG;
    TA0CTL &= ~MC_3;
  }
```

APPENDIX C. IAR EMBEDDED WORKBENCH (EW430) PROJECT SETUP

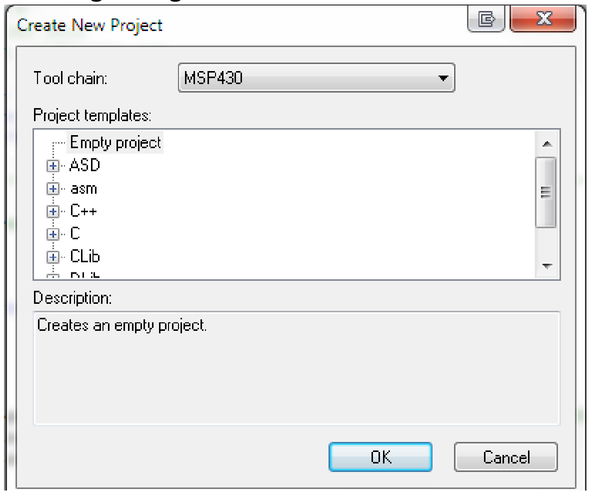
To use the protocol, the following core files are required:

- API package (e.g. API.h, API.c) located in \code\API
- Frame package (e.g. Frame.h, Frame.c) located in \code\DataLink\MAC
- PhyAddress package (e.g. PhyAddress.h, PhyAddress.c) located in \code\DataLink\MAC
- PhyBridge package (e.g. PhyBridge.h) located in \code\DataLink\PhyBridge

A physical bridge implementation must also be selected to build the protocol. The following provides the A110x2500 Physical Bridge as an example. The A110x2500 Physical Bridge requires the following files:

- CC1101 device driver package (e.g. CC1101.h, CC1101.c) located in \code\Physical\A110x2500\Driver
- One module driver package (e.g. A110x2500.h, A110LR09.h, A110LR09.c, A110LR09Config.h) located in \code\Physical\A110x2500\Module\A110LR09
- A110x2500PhyBridge package (e.g. A110x2500PhyBridge.h, A110x2500PhyBridge.c) located in \code\Physical\A110x2500\PhyBridge

In IAR Embedded Workbench, the files may be pulled into a new project. To begin, go to **Project -> Create New Project…**. The following dialog window will be visible.



Creates an empty project.

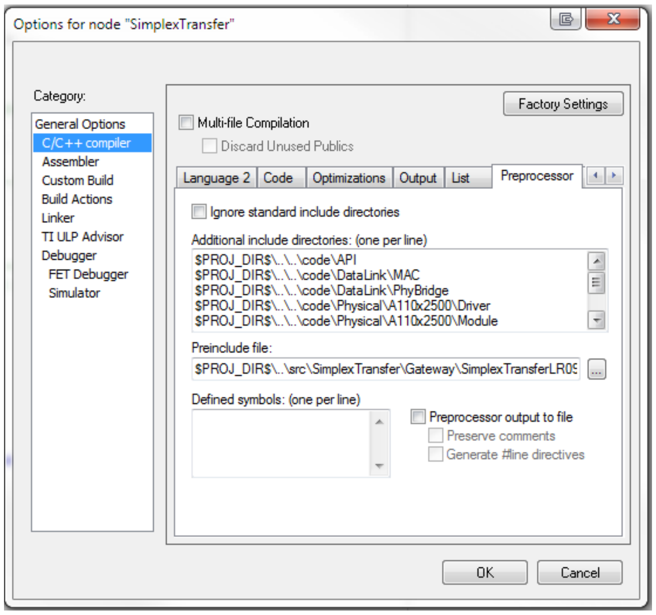
The protocol is written exclusively in C, so either **Empty Project** or **C** project templates will suffice. Once one of the following is selected, click **OK**.

Files may be added to the project by going to **Project->Add Files...**. **Select** all files required for each package (located in **\code\API, \code\DataLink, and \code\Physical**) and **add** them to the project. At this point the protocol layer files have been added to the project. They must now be linked to the project using the **options** menu so that the compiler and linker can have visibility to them.

Go to **Project->Options** and navigate to the **C/C++ compiler** submenu. Next, click on the **Preprocessor** tab to pull up the *Additional include directories* input. At this point, the locations of the protocol directories (e.g. \code\API) should be entered with their location relative to the project directory (an example is shown below).

The following screenshot displays the setup for the example application that is included with the protocol.

Note	Some additional defined symbols are shown, but are only used in the example project to easily transition from one module to another. Please disregard them. The area of interest is the *Additional include directives.*



Multi-file Compilation: ☐
Discard Unused Publics: ☐
Ignore standard include directories: ☐

Additional include directories: (one per line)

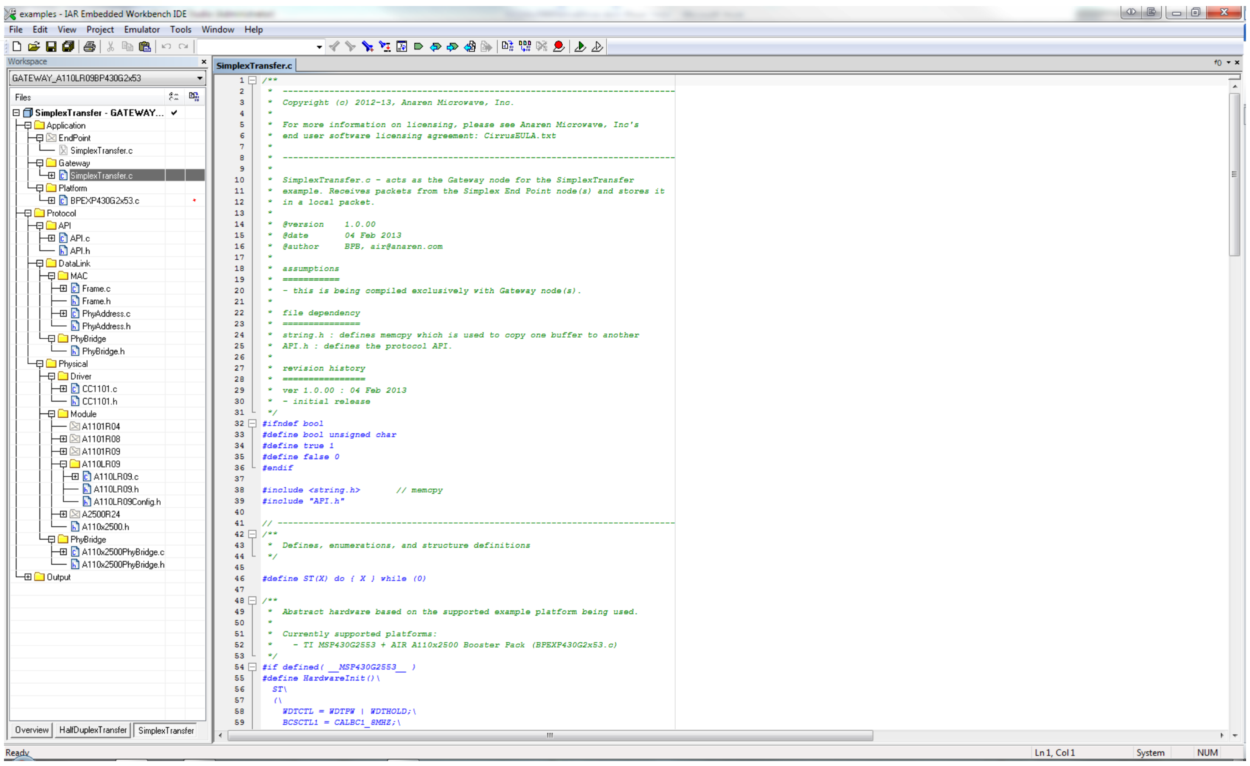
```
$PROJ_DIR$\..\..\code\API
$PROJ_DIR$\..\..\code\DataLink\MAC
$PROJ_DIR$\..\..\code\DataLink\PhyBridge
$PROJ_DIR$\..\..\code\Physical\A110x2500\Driver
$PROJ_DIR$\..\..\code\Physical\A110x2500\Module
```

Preinclude file: `$PROJ_DIR$\..\src\SimplexTransfer\Gateway\SimplexTransferLR0S`

Defined symbols: (one per line)

Preprocessor output to file: ☐
Preserve comments: ☐
Generate #line directives: ☐

The example project, which is included with the protocol (\examples), is shown in its complete form in the following screenshot:



```c
/**
 *  ----------------------------------------------------------------------------
 *  Copyright (c) 2012-13, Anaren Microwave, Inc.
 *
 *  For more information on licensing, please see Anaren Microwave, Inc's
 *  end user software licensing agreement: CirrusEULA.txt
 *
 *  ----------------------------------------------------------------------------
 *
 *  SimplexTransfer.c - acts as the Gateway node for the SimplexTransfer
 *  example. Receives packets from the Simplex End Point node(s) and stores it
 *  in a local packet.
 *
 *  @version    1.0.00
 *  @date       04 Feb 2013
 *  @author     BPB, air@anaren.com
 *
 *  assumptions
 *  ===========
 *  - this is being compiled exclusively with Gateway node(s).
 *
 *  file dependency
 *  ===============
 *  string.h : defines memcpy which is used to copy one buffer to another
 *  API.h : defines the protocol API.
 *
 *  revision history
 *  ================
 *  ver 1.0.00 : 04 Feb 2013
 *  - initial release
 */
#ifndef bool
#define bool unsigned char
#define true 1
#define false 0
#endif

#include <string.h>       // memcpy
#include "API.h"

// ----------------------------------------------------------------------------
/**
 *  Defines, enumerations, and structure definitions
 */

#define ST(X) do { X } while (0)

/**
 *  Abstract hardware based on the supported example platform being used.
 *
 *  Currently supported platforms:
 *    - TI MSP430G2553 + AIR A110x2500 Booster Pack (BPEXP430G2x53.c)
 */
#if defined( __MSP430G2553__ )
#define HardwareInit()\
  ST\
  (\
    WDTCTL = WDTPW | WDTHOLD;\
    BCSCTL1 = CALBC1_8MHZ;\
```

DOCUMENT HISTORY

Date	Author	Description
18 February 2013	BPB	Initial draft