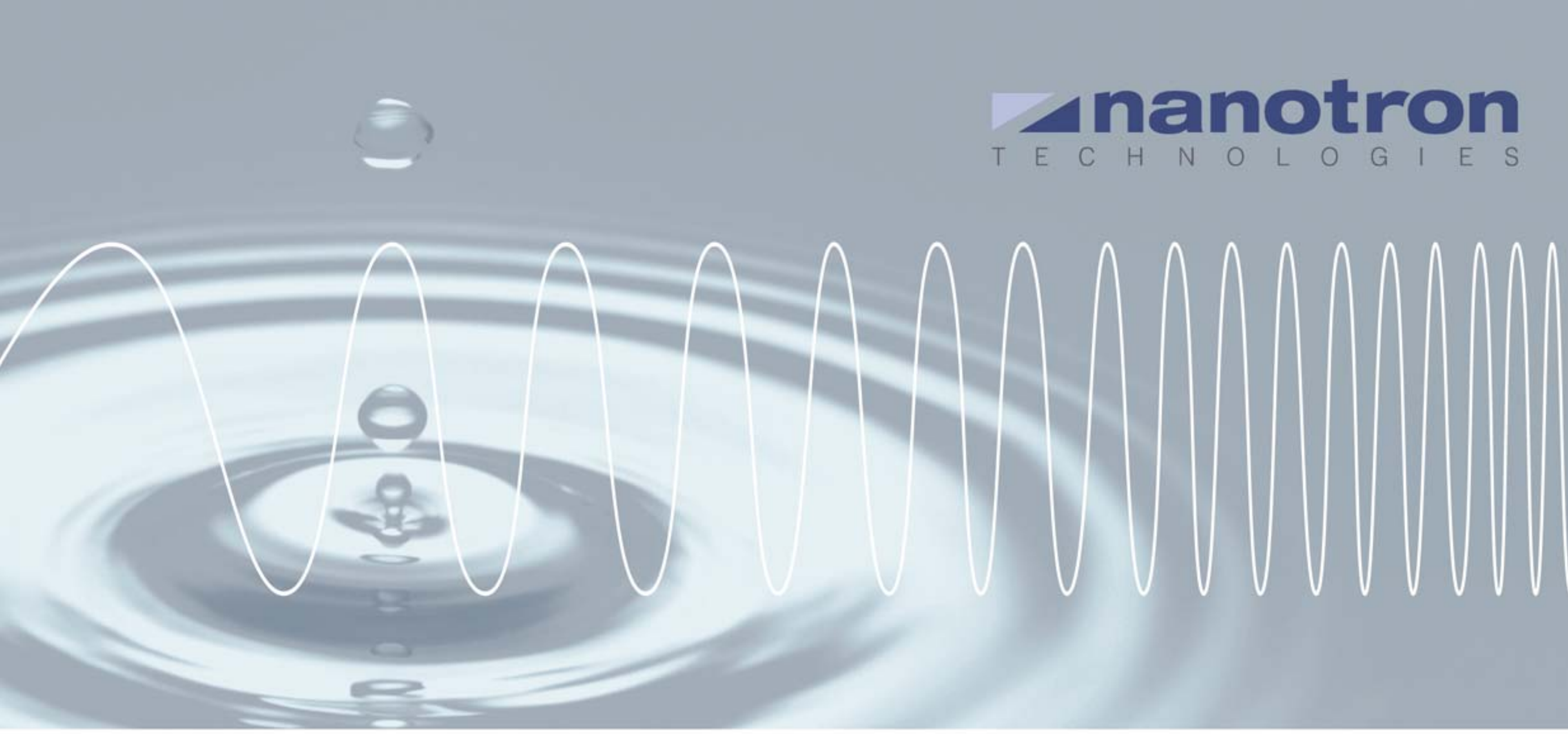

nanoLOC nTRX Driver Suite

User Guide

Version 2.3

NA-07-0240-0407-2.3

Document Information

Document Title:	nanoLOC nTRX Driver Suite User Guide
Document Version:	2.3
Published *(yyyy-mm-dd)*:	2009-04-28
Current Printing:	2009-4-28, 11:48 am
Document ID:	NA-07-0240-0407-2.3
Document Status:	Released

Table of Contents

1 Introduction

This document describes the structure of the *nanoLOC nTRX Driver* model as well as the interface the driver provides for accessing the *nanoLOC TRX Transceiver.* Additionally, a description of the nTRX Driver Demo is provided. This software is a complete sample application for the *nanoLOC DK Board* that is can be compiled and downloaded to the board for evaluation purposes.

1.1 nanoLOC nTRX Driver Overview

Wireless applications that access the *nanoLOC* chip do so through the API of the *nanoLOC nTRX Driver*. Upper layers, including the application layer, communicate with lower layers by sending and receiving messages. This API allows the application layer to access the chip's functions including chip-specific settings and performance criteria such as address matching, error checking, modulation methods, and data transmission rates. Furthermore, hardware adaption layer messages are sent to the *nanoLOC* chip over the SPI interface. The *nanoLOC* API is located within the `phy.c` module of the *nanoLOC nTRX Driver* source code.

1.2 Modules

The *nanoLOC nTRX Driver* consists of the following four modules:

- `ntrxinit.c`

  This module contains the initialization function for the *nanoLOC* chip.
- `ntrxiqpar.c`

  This module contains the chirp sequence values for the chirp sequencer in the *nanoLOC* chip.
- `phy.c`

  This module contains the *nanoLOC nTRX Driver* API and is used for transmitting and receiving messages.
- `ntrxutil.c`

  This module contains the utility functions needed by other *nanoLOC nTRX Driver* modules for calibration and general register access.

The following figure shows the relationship between these four modules:

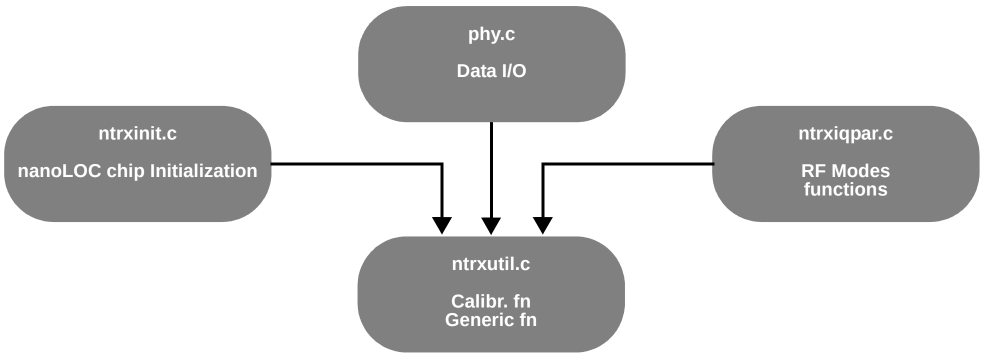

Figure 1: nanoLOC nTRX Driver modules

1.3 Key Settings

Some of the key settings of the *nanoLOC nTRX Driver* include:

- A means for the transceiver to access upper layers
- Read and write values into transceiver registers
- Set the Logical Network ID (which sets the SyncWord)
- Enable/disable address matching
- Enable/disable Broadcast (Brdcast) or Time Beacon (TimeB) packets

- Enable/disable receiver CRC2 checking
- Set CRC2 method to detect transmission bit errors
- Set CSMA/CA (carrier sense method for collision avoidance)
- Enable/disable a backoff scheme for packet collision avoidance
- Enable/disable Automatic Repeat Request (ARQ) on the receiver
- Set the transmission output power
- Calibrate frequency of transmission and reception oscillators
- Set frequency bandwidth
- Select frequency channel (for narrowband transmission)
- Read out range (distance) value
- Manually start and stop packet transmission
- Set data transmission rate
- Enable/disable FEC

1.4 Example Implementation

Figure 2 below shows an example implementation of the *nanoLOC nTRX Driver*.

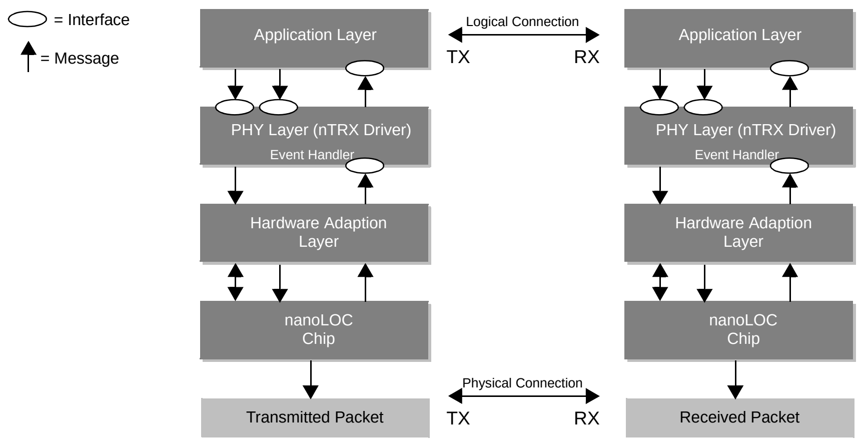

Figure 2: Example nanoLOC TRX Driver implementation

2 nanoLOC nTRX Driver Modules

This section describes the functions of the four modules of the *nanoLOC nTRX Driver.*

2.1 ntrxinit.c – Chip Initialization

2.1.1 Description

The `ntrxinit.c` module contains the initialization function for the *nanoLOC* chip.

This module is used only during startup to set the *nanoLOC* registers with reasonably values. Once this function is called, the *nanoLOC* chip operates in normal mode. These values are only needed during initialization.

However, when using certain power saving modes or RF modes, the *nanoLOC chip* may need to be re-initialized with values that are stored in volatile on-chip memory.

2.1.2 Function

```
bool_t NtrxInit (void);
```

Purpose: This function initializes the *nanoLOC* chip and the *nanoLOC nTRX Driver* to specific preset values. It checks for the consistency of the register table and for correct communication between the microcontroller and the *nanoLOC* chip. It does this by reading the version and revision values of the chip.

2.2 ntrxiqpar.c – Chirp Sequencer

2.2.1 Description

The `ntrxiqpar.c` module contains the chirp sequence values for the *nanoLOC* chip chirp sequencer.

Note: These values should not be modified or altered in any way. They should be taken as is.

This module also contains functions used for setting different RF modes available with the *nanoLOC* chip. When using certain power saving modes, the *nanoLOC* chip may need to be re-initialized with values that are stored in volatile on-chip memory.

Note: The values in the arrays should not be modified or altered in any way. They are fine tuned to provide the best possible and reliable performance and communication quality.

When only a unidirectional communication channel is needed, this module has some memory saving potential.

2.2.2 Functions

```
void NTRXSetRxIQMatrix (uint8_t bandwidth, uint8_t symbolDuration);
```

Purpose: This function sets the IQ-parameter for the receiver part with a chosen bandwidth and symbol duration.

```
void NTRXSetTxIQMatrix (uint8_t bandwidth, uint8_t symbolDuration);
```

Purpose: This function sets the IQ-parameter for the transmitter part with a chosen bandwidth and symbol duration.

```
void NTRXSetAgcValues (uint8_t bandwidth, uint8_t symbolDuration,
    uint8_t symbolRate);
```

Purpose: This function sets the AGC values with a chosen bandwidth, symbol duration, and symbol rate in the *nanoLOC* chip.

`void NTRXSetCorrThreshold (uint8_t bandwidth, uint8_t symbolDuration);`

Purpose: This function sets the correlator threshold values with a chosen bandwidth and symbol duration.

2.3 phy.c – Transmitting and Receiving Messages

The `phy.c` module contains the Application Programming Interface (API) to the *nanoLOC* chip for transmitting and receiving messages.

Note: See *3. nanoLOC API* on page 7 for details.

2.4 ntrxutil.c – Calibration and Register Access

2.4.1 Description

The `ntrxutil.c` module contains utility functions needed by other modules of the *nanoLOC nTRX Driver*. These are:

- Startup functions – used only during startup to initialize the *nanoLOC* chip
- Calibration functions – used for calibration of various parts of the *nanoLOC* chip
- Read and write functions – used for modifying chip settings
- Get and set function – used to ensure other bits at an address are not altered
- Chip configuration functions – used for setting basic chip configuration

2.4.2 Startup Functions

`void NTRXInitShadowRegister (void);`

Purpose: Because not all registers can be read back, it is necessary to backup specific register values in RAM. This function is called during startup to initialize the shadow registers.

`void SetRxStart (void);`

Purpose: This function sets a flag to remember the current state of the chip's receiver. This is used to prevent double starting of the receiver, which is prohibited.

`void SetRxStop (void);`

Purpose: This function is used to access a variable to ensure that the receiver is never started twice. The second call will be silently ignored.

`bool_t NTRXCheckTable (void);`

Purpose: This function is used only to check the integrity of the table and the enumeration list of fields.

2.4.3 Calibration Functions

`void NTRXRxLoAdjust(void);`

Purpose: This function calibrates the values of the switched capacitors according the required RX target frequency of the local oscillator.

`void NTRXTxLoAdjust (void);`

Purpose: This function calibrates the values of the switched capacitors according the required TX target frequency of the local oscillator.

`void NTRXRxLoCalibration (void);`

Purpose: This function calls `NTRXRxLoAdjust` to calibrate the Local Oscillator of the receiver part but enclosed by a stop/start receiver command.

`void NTRXTxLoCalibration (void);`

Purpose: This function calls `NTRXTxLoAdjust` to calibrate the Local Oscillator of the transmitter part but enclosed by a stop/start transmitter command.

`void NTRXAllCalibration (void);`

Purpose: This function calls the functions `NTRXRxLoCalibration` and `NTRXTxLoCalibration` to calibrate the Local RF Oscillators for the transmitter and receiver.

`void NTRXFctCal (void);`

Purpose: This function is for calibrating the frequency of the *nanoLOC* chip. It needs to be called periodically. The time interval depends on the traffic over the air.

2.4.4 Read and Write Functions

These are generic functions for modifying chip settings and are memory optimized. They require the `regCmd` table.

Note: All functions in this section are used to set and to read bits in the registers of the *nanoLOC* chip.

2.4.5 Get and Set Functions

To set or reset certain fields, the following functions ensure that other bits at this address are not altered. Therefore these functions know the address of a field and which bits belong to a individual field. In some cases fields span more than a single byte, as for example the MAC Address which uses six bytes.

2.4.5.1 Setting Upper Address Bits

`void NTRXSetIndexRegister (uint8_t page);`

Purpose: This function handles the upper two address bits of the *nanoLOC* chip. This function is only for compatibility purposes with the *nanoNET TRX Transceiver*. It is recommended that `NTRXSetRamIndex()` be used. For more details about addressing registers, see the *nanoLOC chip (NA5TR1) User Guide*.

`void NTRXSetRamIndex (uint8_t page);`

Purpose: This function handles the upper address bits of the *nanoLOC* chip. For more details about addressing registers, see the *nanoLOC chip (NA5TR1) User Guide*.

2.4.5.2 Modifying Single Byte Register

`void NTRXSetRegister (NTRXCmdTE cmd, uint8_t value);`

Purpose: This function is used to modify single byte registers. It is called with two parameters. This is evaluated first in the function. Then depending on the type of operation, the new register is calculated and finally written to the *nanoLOC* chip.

`void NTRXGetRegister (NTRXCmdTE cmd, uint8_t *value);`

Purpose: This function reads individual bits in registers. Before the read out value is returned to the calling function, the value is masked by the bit mask from the third column from the table.

2.4.5.3 Modifying Multiple Byte Registers

`void NTRXSetNRegister (NTRXCmdTE cmd, uint8_t *value);`

Purpose: This function works with registers spanning multiple bytes in the *nanoLOC* chip. This function is used for the MAC address fields, the payload length, and the actual data.

`void NTRXGetNRegister (NTRXCmdTE cmd, uint8_t *value);`

Purpose: This function reads out registers that span across multiple bytes in the *nanoLOC* chip.

2.4.5.4 Providing Compact Code

`void NTRXProcessTable (const uint8_t *table);`

Purpose: This function processes a variable list of commands to modify chip settings. It is used during initialization and is memory optimized.

```
void NTRXProcessSequence (const uint8_t *table);
```

Purpose: This function is used to efficiently set the register or sequences of data to be written into registers.

2.4.6 nanoLOC Chip Configuration Functions

```
uint8_t NTRXGetTxARQmax (void);
```

Purpose: This function reads out the number of maximum retries for transmission.

```
bool_t NTRXGetRxCRC2mode (void);
```

Purpose: This function obtains the receiver mode, which can be either CRC1 or CRC1 + CRC2.

```
uint8_t NTRXGetRfOutputPower (uint8_t value);
```

Purpose: This function, when set to 0, reads out the output power settings for frame types Data, Broadcast and Time Beacon, and when set to 1, reads out the output power settings for frame types Acknowledgement, Request-to-Send and Clear-to-Send.

```
void NTRXSetStaAddress (uint8_t *address);
```

Purpose: This function sets the source MAC address in the *nanoLOC* chip. This address is used in the MAC header in transmitted messages and for address matching with received messages.

```
void NTRXSetSyncword (uint8_t *value);
```

Purpose: This function sets the Syncword in the *nanoLOC* chip.

Note: All devices need to be set to the same Syncword, otherwise communication is not possible. The bit sequence of the Syncword has to follow certain rules and should not be set randomly. Only Syncwords supplied by *Nanotron* should be used.

```
void NTRXSetupTRxMode (uint8_t fdma, uint8_t symbolDuration, uint8_t
   symbolRate);
```

Purpose: This function is used to set a specific transmission mode in the *nanoLOC* chip.

```
bool_t NTRXCheckVersion (void);
```

Purpose: This function compares the version and revision number of the chip with the version selected during software production and returns the result.

```
uint8_t NTRXGetRfOutputPower (uint8_t value);
```

Purpose: This function reads the selected output power in the *nanoLOC chip*.

```
uint8_t NTRXGetPowerEntry (uint8_t value);
```

Purpose: This function reads the selected output power in the *nanoLOC chip*.

2.5 ntrxutil.c – Calibration and Register Access

```
void NTRXPowerdownMode (uint8_t mode, uint32_t seconds);
```

where:

- `mode` – either be full Powerdown mode (0) or PAD mode (1).
- `seconds` – number of seconds to stay in Powerdown mode.

Purpose: This function both puts the chip in Powerdown mode and sets the duration of Powerdown mode by setting an alarm of the RTC that wakes up the chip after the set time.

3 nanoLOC API

This section describes the Application Programming Interface (API) of the *nanoLOC nTRX Driver.* The API uses the same naming scheme as IEEE 802.15.4 to make application development easier for developers who have experience with 15.4 devices and software.

3.1 General Layer Interface – SAP, MESAP and Callback

All communication between layers is done by sending and receiving messages. Every layer has the same interface and can be easily exchanged or extended without changing the overall interface. Service Access Points (SAPs) are implemented as C functions which accept one parameter in the form of a message. Messages contain all the necessary information required to execute a task from an upper layer.

Figure 3 below shows the general layer interface of the *nTRX Driver*.

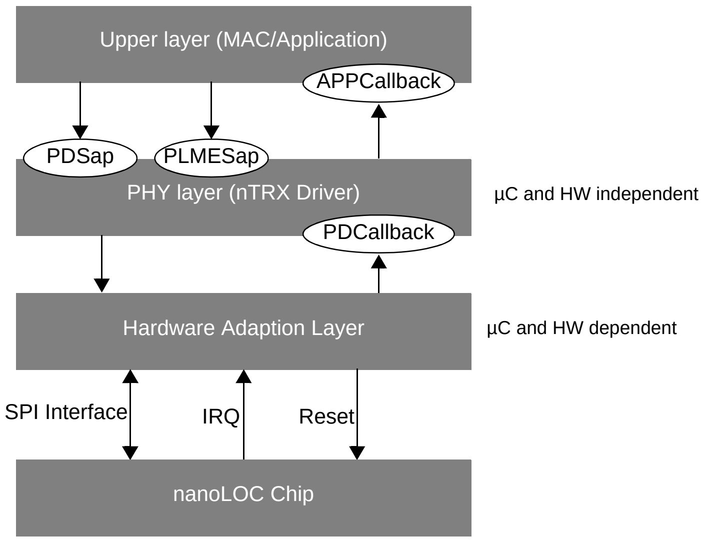

Figure 3: General layer interface

The PHY layer has two Service Access Points (SAPs) for downstream messages generically named

- `LayernameSAP`
- `LayernameMESAP`

It also has one callback function for upstream messages generically named

- `LayernameCallback`

3.2 Service Access Points General Form

3.2.1 Downstream

The general form for downstream Service Access Points (SAPs) are:

```
void LayernameSAP (MsgT *msg);
void LayernameMESAP (MsgT *msg);
```

On the PHY layer, these SAPs are:

```
void PDSap (MsgT *msg)
void PLMESap (MsgT *msg)
```

3.2.2 Upstream

The general form for the upstream Callback function is:

```
void LayernameCallback (MsgT *msg);
```

On the PHY layer, this callback function is:

```
void PDCallback (void)
```

3.2.3 Variable Initialization

Additionally, every layer has variables that have to be initialized. This action is put into a separate function called `LayernameInit`:

```
void LayernameInit (void);
```

On the PHY layer, this function is:

```
void PHYInit (void)
```

3.3 General Purpose Message Structure – MsgT

All information that has to provided to a lower layer is put into a general purpose message structure `MsgT`, which is located in `ntrxtypes.h`. Also, the results or return values are transmitted with the help of this structure.

```
typedef struct
{
      uint8_t prim;
      AddrT addr;
      uint8_t len;
      uint8_t data[128];                 General message structure

      uint8_t status;
      uint16_t value;
      uint8_t attribute;
#     ifdef CONFIG_NTRX_SNIFFER
      uint32_t count;
      AddrT rxAddr;
      uint8_t frameType;                 nanoLOC Packet Sniffer only
      uint8_t extBits;
#     endif /* CONFIG_NTRX_SNIFFER */
} MsgT;
```

The variables in the message structure `MsgT` are described in the following table.

Table 1: Message structure

Variable	Description
uint8_t prim;	Primitive name (1 byte)
AddrT addr;	MAC address of destination station (6 bytes)
uint8_t len;	Length of message (1 byte)
uint8_t data[128];	Payload of message (n bytes to a maximum of 128 bytes)
uint8_t status;	Status indicator used with LayerCallback function (1 byte)
uint16_t value;	Value of requested parameter used for setting or getting chip configuration requests (2 bytes)
uint8_t attribute;	Selected attribute used with the value parameter for setting or getting chip configuration requests (1 byte)

3.4 Hardware Adaption Layer – Microcontroller and Hardware Dependent

The hardware adaption layer is microcontroller and hardware dependent and is used to access the *nanoLOC* chip. It must be written for each microcontroller family. The *nanoLOC Development Kit* uses the *Atmel ATmega128L*, which is part of the *AVR* family of microcontrollers.

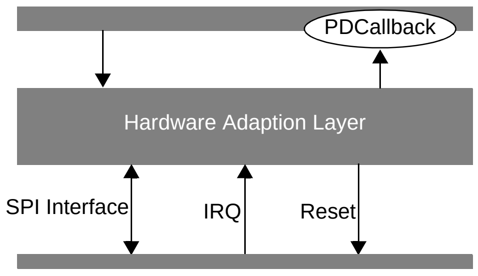

Figure 4: Hardware adaption layer

The implementation of the Hardware Adaption Layer for AVR microcontrollers uses four functions to communicate with the *nanoLOC* chip over the SPI bus as well as an interrupt service routine. The functions for the SPI bus include:

```
void NTRXSPIRead(uint8_t address, uint8_t *buffer, uint8_t len);
void NTRXSPIWrite(uint8_t address, uint8_t *buffer, uint8_t len);
void NTRXSPIReadByte(uint8_t address, uint8_t *buffer);
void NTRXSPIWriteByte(uint8_t address, uint8_t buffer);
```

where:

- `NTRXSPIRead` reads multiple bytes to the *nanoLOC* chip
- `NTRXSPIWrite` writes multiple bytes to the *nanoLOC* chip
- `NTRXSPIReadByte` reads single bytes to the nanoLOC chip
- `NTRXSPIWriteByte` writes single bytes to the nanoLOC chip

`NTRXSPIReadByte` and `NTRXSPIWriteByte` have been added for speed optimization as they are executed faster than `NTRXSPIRead` and `NTRXSPIWrite`.

The function used for interrupt service routines is:

```
void NTRXInterrupt (void)
```

The interrupt service routine will read the IRQ status register and raise a flag for transmission finished or message reception. The actual processing of the interrupt source is done in normal context through the use of the function `PHYPoll()`.

3.5 PHY Layer – nTRX Driver

The PHY layer provides two downstream SAPs and an upstream callback function.

- Downstream SAPs:

```
void PDSap (MsgT *msg)
void PLMESap (MsgT *msg)
```

`PDSap` is provided for data transmission while `PLMESap` is a Physical Layer Management Entity SAP and is used for *nanoLOC* chip configuration requests and queries.

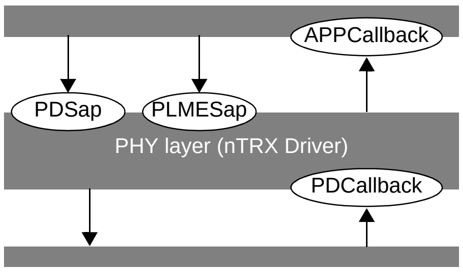

Figure 5: PHY layer

- Upstream Callback Function:

```
void PDCallback (void)
```

3.6 Chip Initialization

To prepare the *nanoLOC* chip for message sending and receiving, the following function is used:

```
bool_t NtrxInit(void)
```

3.7 Data Communication

This service requires two functions, one for downstream (transmission) and one for callback (reception):

Downstream SAP:

```
void PDSap (MsgT *msg)
```

Callback function:

```
void PDCallback (void)
```

The structure `MsgT` is used with the downstream SAP `PDSap` using the following parameters:

```
typedef struct
{
      uint8_t prim;
      AddrT addr;
      uint8_t len;
      uint8_t data[128];
      uint8_t status;     }
      uint16_t value;     } Not used for sending messages
      uint8_t attribute   }
} MsgT;
```

One of three primitives are used to indicate the requested service:

- message transmission
- message reception
- hardware acknowledgement

Message Transmissions

`PD_DATA_REQUEST`

Purpose: Indicates that a payload contained in this message is to be transmitted to a given destination.

Mandatory parameters are `prim`, `data`, `len`, and `addr`.

Message Reception

`PD_DATA_INDICATION`

Purpose: Indicates that a message has been received from another device.

Valid parameters are `data`, `len`, and `addr`.

Hardware Acknowledgement

`PD_DATA_CONFIRM`

Purpose: Indicates that a `PD_DATA_REQUEST` message was successfully executed.

This message includes the original message payload (data) and destination address, as well as the length of payload. The parameter `status` indicates the success or failure of the request.

3.8 Application Layer

The application layer accesses two downstream SAPs on lower layers and provides to lower layers a callback function.

In the *nanoLOC nTRX Driver*, `PDSap()` on the PHY layer provides to the application layer a SAP for data transmission and reception, while `PLMESap()` on the PHY layer provides to the application layer a SAP for chip configuration and PHY layer configuration.

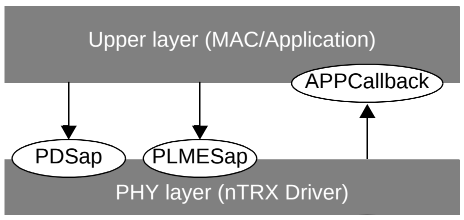

Figure 6: Application layer

This layer uses the same general message structure as the PHY layer, which is located in `ntrxtypes.h`.

```
typedef struct
{
      uint8_t prim;
      AddrT addr;
      uint8_t len;
      uint8_t data[128];
      uint8_t status;
      uint16_t value;
      uint8_t attribute
} MsgT;
```

The callback function used in the application layer is:

```
void APPCallback (MsgT *msg)
```

3.9 Chip Configuration

Chip configuration uses the SAP `PLMESap`:

```
void PLMESap (MsgT *msg);
```

The the SAP `PLMESap` uses the structure `MsgT`. The parameter `msg` of type `MsgT` uses the following structure elements to perform configuration operations:

- `prim`
- `data[128]` or `value`
- `attribute`

This structure is shown below.

```
typedef struct
{
      uint8_t prim;
      AddrT addr;          } Not used
      uint8_t len;         }
      uint8_t data[128];
      uint8_t status;   Not used
      uint16_t value;
      uint8_t attribute
} MsgT;
```

prim

The structure element `prim` distinguishes between two configuration read and write operations from a calling function:

- `PLME_GET_REQUEST`
- `PLME_SET_REQUEST`

The SAP function `PLMESap` replies from the calling function using the following primitives:

- `PLME_SET_CONFIRM`
- `PLME_GET_CONFIRM`

attribute

The structure element `attribute` select a particular parameter that is to be modified (set the value or retrieve contents. For example, obtaining the current MAC address or setting the MAC address.

value

The structure element `value` is used when the return value is a simple 16 bit variable.

data[128]

The structure element `data[128]` is used for larger or more complex values.

3.9.1 Code Example

The following code example demonstrates switching on the receiver of the nanoLOC chip and then obtains the current MAC address set in the chip.

1 Switch on the receiver of the *nanoLOC* chip.

```
      msg.prim = PLME_SET_REQUEST;
      msg.attribute = PHY_RX_CMD;
      msg.value = PHY_TRX_OFF;
      PLMESap (&msg);
```

2 Obtain the currently set MAC address

```
msg.prim = PLME_GET_REQUEST;
msg.attribute = PHY_MAC_ADDRESS1;
PLMESap (&msg);
/* MAC address is now in data[] */
/* msg.prim is set to PLME_GET_CONFIRM */
```

3.9.2 Service Primitives

One of three primitives are used to distinguish the requested service:

`PLME_GET_REQUEST`

Purpose: Queries a setting of the layer parameter. For example, is FEC set to on?

The `attribute` parameter is used to select one of the supported attributes (see *Supported Attributes* on page 13), while the `value` parameter selects the current setting of the attribute. The `value` field is only valid if the status field indicates no error.

`PLME_SET_REQUEST`

Purpose: Sets a parameter of the layer. For example, set FEC to on.

The `attribute` parameter is used to select one of the supported attributes (see *Supported Attributes* on page 13), while the `value` parameter provides the new value to set.

`PLME_SET_CONFIRM`

Purpose: Returns the status in the `status` parameter of the previous request. For example, FEC is set to on.

3.9.3 Supported Attributes

The following attributes are available to be set or queried through the function `PLMESap()`.

`PHY_LOG_CHANNEL`

Purpose: Selects or requests a logical channel, which is a predefined set of transmission mode settings. The following logical channels are supported:

Table 2: NTRX_LogChannel attributes

Logical Channel	Bandwidth [MHz]	Bit rate [ns]	FEC	Centerfreq ID
NLC_80_1_N_0	80	1000	No	0
NLC_22_4_N_1	22	4000	No	1
NLC_22_4_N_7	22	4000	No	7
NLC_22_4_N_13	22	4000	No	13

`PHY_CHANNEL`

Purpose: Sets the center frequency used in the `NTRX_Mode`. The following center frequencies can be selected:

Table 3: NTRX_Channel attributes

Channel ID	Center Frequency [GHz]	Description
0	2.442.175	CSS, 80 MHz
1	2.412	802.11b, CHNL_ID 1
2	2.417	802.11b, CHNL_ID 2

Table 3: NTRX_Channel attributes (Continued)

Channel ID	Center Frequency [GHz]	Description
3	2.422	802.11b, CHNL_ID 3
4	2.427	802.11b, CHNL_ID 4
5	2.432	802.11b, CHNL_ID 5
6	2.437	802.11b, CHNL_ID 6
7	2.442	802.11b, CHNL_ID 7
8	2.447	802.11b, CHNL_ID 8
9	2.452	802.11b, CHNL_ID 9
10	2.457	802.11b, CHNL_ID 10
11	2.462	802.11b, CHNL_ID 11
12	2.467	802.11b, CHNL_ID 12
13	2.472	802.11b, CHNL_ID 13
14	2.484	802.11b, CHNL_ID 14 (Japan)
15	2412.75	HR low band center frequency
16	2470.76	HR high band center frequency

`PHY_ARQ`

Purpose: Turns on and off automatic retransmissions. When on, it also sets the maximum number of retransmissions. The range for maximum retransmissions is 0 to 14.

`PHY_FEC`

Purpose: This attribute turns on and off Forward Error Correction (FEC). For more details, see *nanoLOC TRX (NA5TR1) Transceiver User Guide*.

`PHY_TX_POWER`

Purpose: This attribute sets the output power of the transmitter. The programmable output power of the *nanoLOC* chip is from -33 dBm to 0 dBm.

`PHY_ADDR_MATCHING`

Purpose: This attribute turns on and off address matching. For more details, see *nanoLOC TRX (NA5TR1) Transceiver User Guide*.

`NTRX_RTC`

Purpose: This attribute accesses the Real Time Clock of the *nanoLOC* chip. For more details, see *nanoLOC TRX (NA5TR1) Transceiver User Guide*.

`PHY_PWR_DOWN_MODE`

Purpose: This attribute is used to put the *nanoLOC* chip into PowerDownModeFull or PowerDownModePad. For more details, see *nanoLOC TRX (NA5TR1) Transceiver User Guide*.

`PHY_RECALIBRATION`

Purpose: This attribute is used to switch cyclic recalibration on or off. In case of on, it also sets the calibration interval. For more details, see *nanoLOC TRX (NA5TR1) Transceiver User Guide*.

`PHY_MAC_ADDRESS1`

Purpose: This attribute is used to set the local MAC address. This address is used for address matching and as sender address when transmitting messages. For more details, see *nanoLOC TRX (NA5TR1) Transceiver User Guide*.

The application will now be operating in the newly selected mode.

4 nanoLOC Ranging Modes

This section describes *nanoLOC's* two ranging modes – normal mode and fast mode. Ranging states and ranging functions are also described. The ranging functions are part of the PHY layer in the *nanoLOC nTRX Driver*.

4.1 Propagation and Processing Delay

nanoLOC ranging uses two types of transmissions, Data packets and hardware Acknowledgments, to obtain the following two types of time measurements:

- **TX Propagation Delay**, which is the time for a data or acknowledgement packet to be transmitted from one station to another. As the speed of a signal propagating through the air is known (the speed of light), the time in which a packet is sent from one station to another can be used to calculate the distance between the stations.
- **Processing Delay**, which is the time required to process a received data packet and generate and transmit a hardware acknowledgement packet to the sending station. This also is a known value and is used as part of the ranging calculations.

After these time measurements are accumulated, a ranging formula used to obtain a ranging distance between the two *nanoLOC* nodes.

4.2 Ranging Modes

Two ranging modes are selectable in the *nanoLOC* chip: normal ranging mode and fast ranging mode.

4.2.1 Normal Ranging Mode

Normal ranging mode uses a ranging methodology developed by Nanotron called *Symmetrical Double-Sided Two-Way Ranging* (SDS-TWR):[1]

- This ranging methodology is *symmetrical* in that the measurement from the local *nanoLOC* station to the remote *nanoLOC* station is mirrored by a measurement from the remote *nanoLOC* station to the local *nanoLOC* station (ABA to BAB).
- It is *Double-Sided* in that only two stations (a remote and a local station) are used for ranging measurements.
- It is *Two-Way* in that a data packet is sent from one station and a hardware acknowledgement is automatically sent back to the sending station.

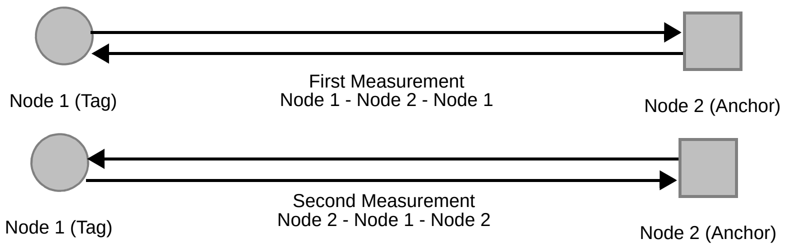

Figure 7: Normal ranging mode

Figure 8 below illustrates normal ranging mode using SDS-TWR.

1. For more information about SDS-TWR, see the white paper *Real Time Location Systems*.

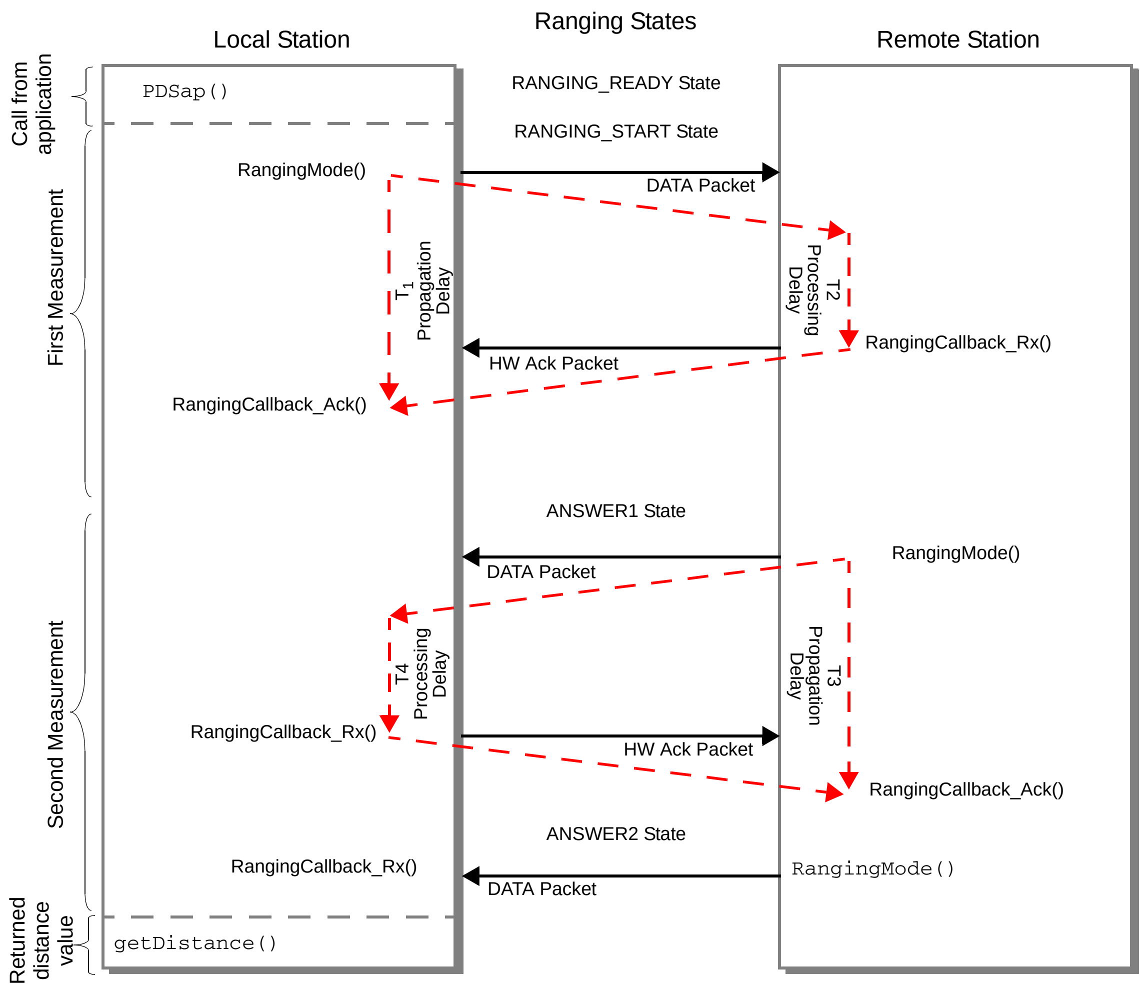

Figure 8: Normal ranging mode using SDS-TWR

Note: An example time to obtain a ranging value in normal ranging mode between a local and a remote station at 80 MHz and 1000 ns is 6 ms.

4.2.2 Fast Ranging Mode

Fast ranging mode uses the same ranging methodology as normal ranging mode, except that it is not symmetrical; that is, only one set of measurements are used (ABA).

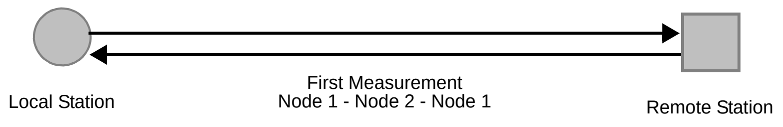

Figure 9: Fast ranging mode

Figure 10 below illustrates fast ranging mode using half SDS-TWR.

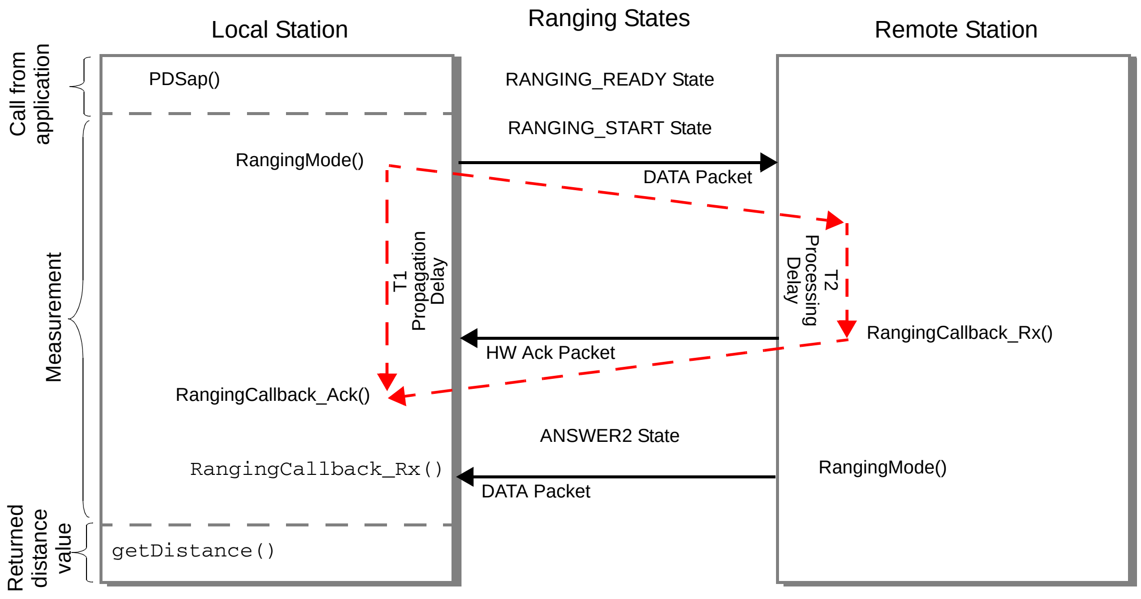

Figure 10: Fast ranging mode using half SDS-TWR

Note: An example time to obtain a ranging value in fast ranging mode between a local and a remote station at 80 MHz and 1000 ns is 3.8 ms.

4.3 Ranging Formulas

In normal ranging mode, a ranging measurement between a local station and a remote station is obtained using the following formula:

$$\text{Distance} = \frac{(T_1 - T_2) + (T_3 - T_4)}{4}$$

where:

T_1 - T_2 = `Answer1`

T_1 = propagation delay time of a round trip between a local and a remote station

T_2 = processing delay in the remote station

T_3 - T_4 = `Answer2`

T_3 = propagation delay time of a round trip between a remote and a local station

T_4 = processing delay in the local station

In fast ranging mode, a ranging measurement between a local and a remote station is obtained using the following formula

$$\text{Distance} = \frac{(T_1 - T_2)}{2}$$

where:

T_1 - T_2 = `Answer1` = `Answer2`

T_1 = propagation delay time of a round trip between a local and a remote station

T_2 = processing delay in the remote station

4.4 Requesting Ranging Service

Both normal and fast ranging modes are done automatically within the PHY layer (*nanoLOC nTRX Driver)* through the selection of primitives in a message structure. This way the application can be performing normal wireless communication and then request ranging when required. For example, a sensor could be transmitting temperature readings and when requested send location coordinates of the sensor.

An application requests the ranging service by calling `PDSap()` with a message structure `MsgT` containing a ranging request primitive and the MAC address of the destination station:

```
void PDSap (MsgT *msg)
```

4.4.1 Requesting Normal Ranging Mode

For normal ranging, the structure `MsgT` is set with the following variables:

Table 4: Message structure for normal ranging

Variable	Description
uint8_t prim;	PD_RANGING_REQUEST
AddrT addr;	MAC address of destination address which is the ranging target.

4.4.2 Requesting Fast Ranging Mode

For fast ranging, `MsgT` is set with the following variables:

Table 5: Message structure for fast ranging

Variable	Description
uint8_t prim;	`PD_RANGING_FAST_REQUEST`
AddrT addr;	MAC address of destination address which is the ranging target.

4.5 Obtaining the Ranging Measurement

To obtain the ranging distance, T_1, T_2, T_3, and T_4 (for normal ranging mode) or T_1, T_2 (for fast ranging mode) are provided to the local station. As was seen previously, normal ranging mode has two ranging cycles `Answer1` and `Answer2`, while fast ranging mode has one ranging cycle `Answer1` which is read as `Answer2`. These two cycles are described in the following sections.

4.5.1 First Ranging Cycle – Answer1

The first ranging cycle involves obtaining times T_1 and T_2.

Obtaining Time T_1 – `RANGING_START` state

After the application calls `PDSap()` with a ranging primitive, the PHY layer calls `rangingmode()` to start ranging with the remote station with set the MAC address.

`RangingMode()` on the local station sends a data packet to the remote station which automatically returns a hardware acknowledgement to the local station. The round trip time (ABA) is the time value T_1, which is the propagation delay of the data packet, the processing delay in the remote station, and the propagation delay of the hardware acknowledgment.

`Ranging_Start` state is shown in Figure 11 below.

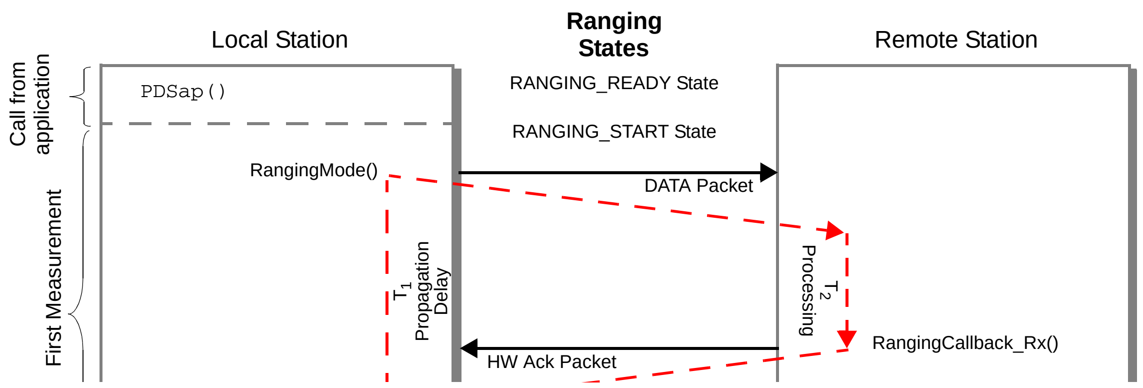

Figure 11: Ranging_Start state

Obtaining Time T_2 – `ANSWER1` State

T_2, which is the processing delay in the remote station during the first round trip time, needs to be sent to the local station. The function `RangingMode()` sends a Data packet to the local station with the time T_2 as the data payload.

By sending this Data packet, the local station obtains time T_2, which concludes the first ranging cycle, but also initiates the second ranging cycle (normal ranging mode only) by sending a hardware acknowledgement to the remote station.

Note: As fast ranging mode has only one ranging cycle, `Answer1` state becomes `Answer2` state which concludes the ranging measurements a ranging distance is generated.

`Answer1` state is shown in Figure 12 below.

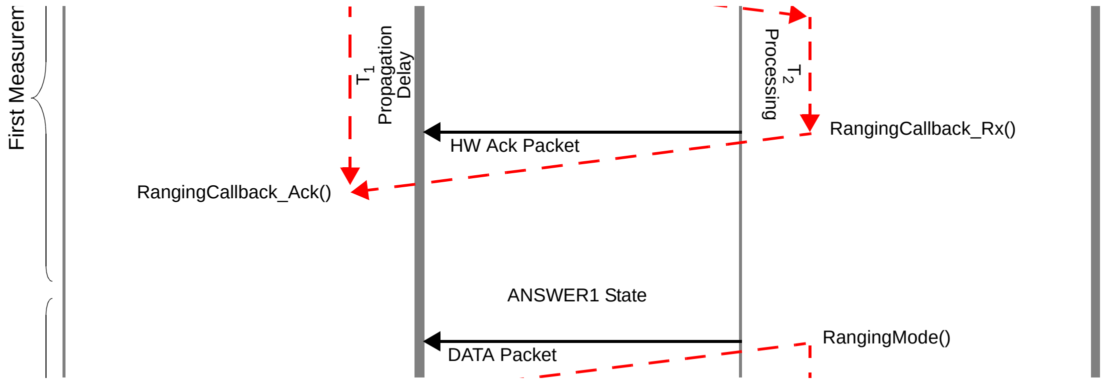

Figure 12: Answer1 state

4.5.2 Second Ranging Cycle – Answer2 State

Obtaining Time T_3 – `ANSWER2` State

The hardware acknowledgement automatically sent back to the remote station causes the time T_3 to be generated. The round trip time T_3 is the propagation delay of the data packet, the processing delay in the remote station, and the propagation delay of the hardware acknowledgment.

`Answer2` state is shown in Figure 13 below.

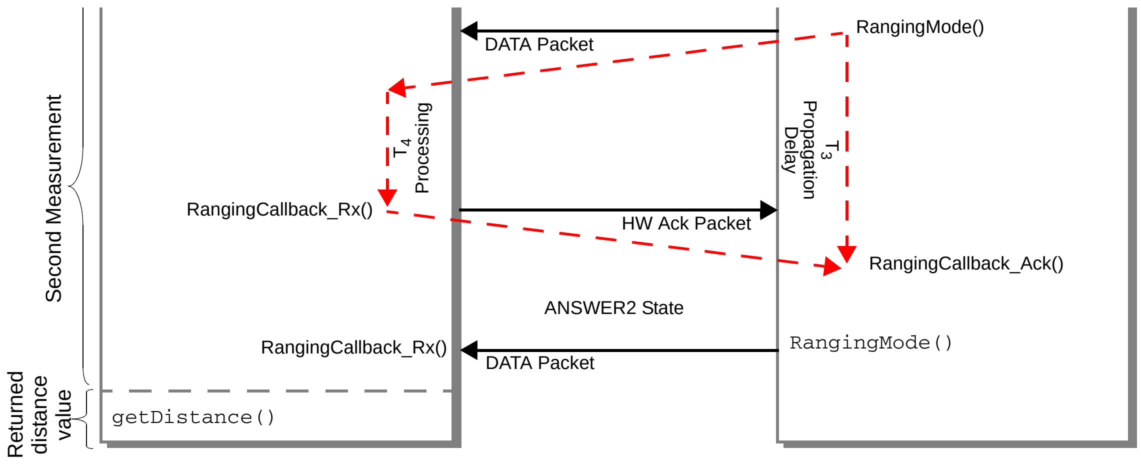

Figure 13: Answer2 state

A final Data packet is sent to the local station needs to obtain the time value T_3 from the remote station. The local station generates a hardware acknowledgement, which is ignored.

Obtaining Time T_4

The final time required T_4, which is the processing delay on the local station, is provided along with all other time values to the function `getDistance()`.

Calculating Time T_4 is shown in Figure 14 below.

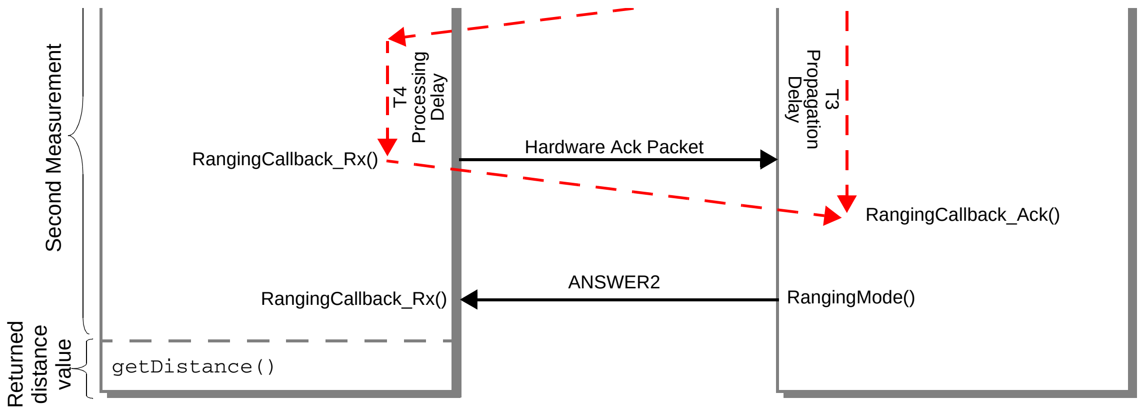

Figure 14: Answer2 state to ranging_ready state

Calculating the Ranging Distance

With all four time values, T_1, T_2, T_3, and T_4, provided to the local station (or two time values T_1, T_2 in the case of fast ranging mode), the function `getDistance()` calculates the ranging distance using the ranging formulas described earlier.

4.5.3 Success and Error Messages

After the ranging service is requested, a message will be received from the PHY layer with either a success response or a failure response. In the case of success, the PHY layer will provide the measured distance in meters. In the case of error, a status indication will provide the cause of the error.

Success Messages – Normal Ranging Mode

For normal ranging mode, the PHY layer sends to the application layer a message with the ranging result having a message structure containing the following information:

Table 6: Message structure for callback

Variable	Description
uint8_t prim;	PD_RANGING_INDICATION
AddrT addr;	MAC address of destination address which is the ranging target.
uint8_t len;	Length of data in the `data` variable.
uint8_t data[128];	Distance value (from `getDistance()`) and error status
uint8_t status;	Not required for `PD_RANGING_INDICATION`
uint16_t value;	Not required for `PD_RANGING_INDICATION`
uint8_t attribute;	Not required for `PD_RANGING_INDICATION`

Success Messages – Fast Ranging Mode

For fast ranging, the PHY layer will send to the application layer a message with the ranging result with the message structure containing the following information:

Table 7: Message structure for callback

Variable	Description
uint8_t prim;	PD_RANGING_FAST_INDICATION
AddrT addr;	MAC address of destination address which is the ranging target.
uint8_t len;	Length of data in the `data` variable.
uint8_t data[128];	Distance value (from `getDistance()`) and error status
uint8_t status;	Not required for `PD_RANGING_FAST_INDICATION`
uint16_t value;	Not required for `PD_RANGING_FAST_INDICATION`
uint8_t attribute;	Not required for `PD_RANGING_FAST_INDICATION`

Error Messages

If an error condition occurs, the PHY layer will send to the application layer a message with the message structure containing an error status.

- For normal ranging mode:

Table 8: Message structure for callback

Variable	Description
uint8_t prim;	PD_RANGING_INDICATION
AddrT addr;	MAC address of destination address which is the ranging target.
uint8_t len;	Length of data in the `data` variable.
uint8_t data[128];	Distance value (from `getDistance()`) and error status
uint8_t status;	Not required for `PD_RANGING_FAST_INDICATION`
uint16_t value;	Not required for `PD_RANGING_FAST_INDICATION`
uint8_t attribute;	Not required for `PD_RANGING_FAST_INDICATION`

- For fast ranging mode:

Table 9: Message structure for callback

Variable	Description
uint8_t prim;	PD_RANGING_FAST_INDICATION
AddrT addr;	MAC address of destination address which is the ranging target.
uint8_t len;	Length of data in the `data` variable.
uint8_t data[128];	Distance value (from `getDistance()`) and error status
uint8_t status;	Not required for `PD_RANGING_FAST_INDICATION`
uint16_t value;	Not required for `PD_RANGING_FAST_INDICATION`
uint8_t attribute;	Not required for `PD_RANGING_FAST_INDICATION`

Possible error messages include:

- `STAT_NO_ERROR` – success
- `STAT_NO_REMOTE_STATION` – no hardware acknowledge received
- `STAT_NO_ANSWER1` – no `Answer1` received
- `STAT_NO_ANSWER2` – no `Answer2` received
- `STAT_PACKET_ERROR_TX` – no hardware acknowledgement received on remote station
- `STAT_PACKET_ERROR_RX1` – no `Answer1` received on remote station
- `STAT_PACKET_ERROR_RX2` – no `Answer2` received on remote station
- `STAT_RANGING_VALUE_ERROR` – ranging value not logical (value < 0)

5 Driver Demo – Hardware Access/Interface Modules

This section describes additional modules that are required for generating a full working demo application. To generate this application, additional modules are required for accessing the hardware on the *nanoLOC DK Board* and for providing functions to use certain interfaces on the board. These modules are described in the following sections.

5.1 appl.c Module – User Application

5.1.1 Description

The `appl.c` module contains the user application. In this example it uses the serial interface to build a wireless link between two terminals to transmit and receive ASCII (printable) characters.

5.1.2 Functions

```
void ApplCallback (uint8_t *addr, uint8_t *payload,
   uint16_t length);
```

Purpose: This function receives every incoming message and sends the payload to the serial interface.

```
void InitApplication (void);
```

Purpose: This function initializes variables used in the application, such as addresses and buffers.

```
void IsAlive (void);
```

Purpose: This function indicates the operational state of the application through a blinking LED.

```
void PollApplication (void);
```

Purpose: This function handles all user input and provides the configuration menu.

5.2 hwclock.c Module – Timer

5.2.1 Description

The `hwclock.c` module handles the timer used for the indication LEDs on the *nanoLOC DK Board.*

Functions

```
void hwclock_init (void);
```

Purpose: This function initializes the clock in the microcontroller.

```
uint32_t hwclock (void);
```

Purpose: This function returns the time in milliseconds since power-on.

```
SIGNAL (SIG_OVERFLOW0);
```

Purpose: Interrupt service routine that is called every 10 ms to update the time-tick counter. It also monitors the keys on the *nanoLOC DK Board*.

Note: `SIGNAL` is a reserved word for interrupt service routines handling interrupts generated by an *ATMEL AVR* microcontroller. For more information, refer to *ATMEL AVR User Guide*.

5.3 usart.c – Serial Line

5.3.1 Description

The `usart.c` module handles the serial line on the *nanoLOC DK Board*.

5.3.2 Functions

```
void console_init (void);
```

Purpose: This function initializes the serial port of the microcontroller and redirects the stream for input, output, and error messages.

```
SIGNAL(SIG_USART1_RECV);
```

Purpose: This function is the interrupt service routine to handle incoming data from the serial interface.

```
int _getchar(FILE *stream);
```

Purpose: This function has to be provided for library I/O functions for default input.

```
int _putchar(char c, FILE *stream);
```

Purpose: This function has to be provided for library I/O functions for default output.

```
int kbhit (void);
```

Purpose: This function checks the input buffer of the serial interface for new data.

```
char *read_line (char buf[]);
```

Purpose: This function collects all incoming data from the serial line. Additionally, this function handles backspaces and carriage return / linefeed conversion.

5.4 nnspi.c Module – SPI Communication

5.4.1 Description

The `nnspi.c` module contains the hardware driver for the SPI communication between the AVR microcontroller and the *nanoLOC* chip on the *nanoLOC DK Board*.

5.4.2 Function

```
void NanoReset (void);
```

Purpose: This function generates a defined reset impulse for the *nanoLOC* chip.

```
void InitSPI (void);
```

Purpose: This function initializes the SPI interface of the microcontroller.

```
void SetupSPI (void);
```

Purpose: This function configures the SPI interface to work in the same mode as the *nanoLOC* chip.

```
void NTRXReadSPI (uint8_t address, uint8_t *buffer,
   uint8_t length);
```

Purpose: This function reads [`length`] bytes into [`buffer`] starting at [`address`].

```
void NTRXWriteSPI (uint8_t address, uint8_t *buffer, uint8_t length);
```

Purpose: This function writes [`length`] bytes from [`buffer`] into *nanoLOC* registers starting at [`address`].

```
void nanosetIRQ (bool_t value);
```

Purpose: This function enables or disables the interrupt for the *nanoLOC* chip.

5.5 main.c Module – Main Loop

5.5.1 Description

The `main.c` module contains the startup code and the main control loop for the application.

5.5.2 Starting Up and Running an Application

To start up and run an application, the `main` function performs the following:

1 The hardware, timer, serial interface and I/O ports are initialized.

The procedure for initializing the hardware and I/O ports are dependent on the hardware and is, therefore, not described here.

2 The SPI interface is initialized to communicate with the *nanoLOC chip*.

The procedure for initializing the SPI interface is depended on the user hardware and is, therefore, not described here

3 The nanoLOC chip is initialized.

See "*Initializing the nanoLOC TRX Transceiver* on page 27

4 The user application is set up.

See *Setting Up the User Application* on page 27

5 The main loop is entered to check for user requests, receive data from the *nanoLOC* chip or update the LEDs for indicating different events.

See *The Main Loop* on page 27.

5.5.3 Initializing the nanoLOC TRX Transceiver

To initialize the *nanoLOC chip*, do the following:

1 Call the following function to correctly initialize all registers for normal 80 MHz, 1 Mbit/s operation, and for starting the receiver:

```
void NTRXInit(void);
```

2 To allow an application to use the *nanoLOC* chip, the application must then set the MAC address.

Note: In this example the *nanoLOC* chip is not interrupt driven.

5.5.4 Setting Up the User Application

To set up the user application, do the following:

1 Initialize the application using the following function:

```
void InitApplication (void);
```

2 Set the MAC address of *nanoLOC* chip for address matching and as sender address for transmitted messages.

3 Additional settings as required should be made at this point.

5.5.5 The Main Loop

The main loop periodically checks for events generated by the software (such as by application requests) or by external events (such as by the *nanoLOC* chip*)*. Timer events are interrupt driven and handled by an interrupt service routine (ISR).

1 The following function is the user application and entry point for processing transmitted and received data:

```
void PollApplication (void);
```

2 The following function checks for incoming messages from the *nanoLOC* chip and delivers the received data to the application:

```
void NTRXUpdate (void);
```

3 The following function is only used to provide visual feedback to the developer:

```
voidIsAlive (void);
```

5.5.6 Transmitting Messages

To transmit a message it is necessary to do the following steps:

1 Set the MAC header in the transmitter.

2 Write the payload to the transmit buffer in the *nanoLOC* chip.

3 Start the transmitter.

4 Send the message using the following function.

```
void NTRXSendMessage (uint8_t *addr, PtrT payload, uint16_t length)
```

where:

- `addr` is the address of the recipient of the message
- `payload` is a pointer to the data buffer containing the payload
- `length` is the length of the payload in bytes

NTRXSendMessage calls the following three functions in succession:

- `NTRXTxHeader (addr, length, rsvd1, rsvd2, rsvd3);`
- `NTRXTxData (payload, length);`
- `NTRXTxStart ();`

5.5.7 Receiving Messages

When a message has been successfully received, the `NTRXUpdate` function is used to read out the header data and payload from the *nanoLOC* chip using the following functions:

```
void NTRXRxMacHeader (void);
```

and

```
void NTRXRxReceive (void);
```

After retrieving all necessary data, the receiver is restarted in order to be ready for the next message. The received data is then delivered to the upper layer. In this case the upper layer is the user application itself.

Index

Revision History

Version	Date	Description/Changes
1.00	2007-03-15	Initial version.
1.1	2007-11-09	minor textual edits; NTRXSetTxDestAddress function no longer included; new parameter added to NTRXSendMessage; NTRXPowerdownMode function added; addr parameter added to ApplCallback function; iohigh and iolow modules replaced by usart.c module which handles the serial line on the nanoLOC DK Board; in section 3.5.6 destination address now set by a parameter in NTRXSendMessage function; description of ranging functions added.
2.0	2007-12-12	nanoLOC API section updated; nanoLOC Ranging section updated; minor text edits throughout.
2.1	2008-04-07	Updated template, minor text changes.
2.2	2008-08-08	Index added; chip configuration details and code example added; general edit throughout.
2.3	2009-04-28	The following data type names where modified: MyByte8T -> uint8_t MyAddrT -> AddrT MyWord16T -> uint16_t MyBoolT -> bool_t MyMsgT -> MsgT MyDword32T ->uint32_t

About Nanotron Technologies GmbH

Nanotron Technologies GmbH develops world-class wireless products for demanding applications based on its patented Chirp transmission system - an innovation that guarantees high robustness, optimal use of the available bandwidth, and low energy consumption. Since the beginning of 2005, Nanotron's Chirp technology has been a part of the IEEE 802.15.4a draft standard for wireless PANs which require extremely robust communication and low power consumption.

ICs and RF modules include *nanoNET TRX Transceiver*, *nanoLOC TRX Transceiver*, and ready-to-use or custom wireless solutions. These include, but are not limited to, industrial monitoring and control applications, medical applications (Active RFID), security applications, and Real Time Location Systems (RTLS). *nanoNET* and *nanoLOC* are certified in Europe, United States, and Japan and supplied to customers worldwide.

Headquartered in Berlin, Germany, *Nanotron Technologies GmbH* was founded in 1991 and is an active member of IEEE and the ZigBee alliance.

Further Information

For more information about this product and other products from Nanotron Technologies, contact a sales representative at the following address:

Nanotron Technologies GmbH
Alt-Moabit 60
10555 Berlin, Germany
Phone: +49 30 399 954 - 0
Fax: +49 30 399 954 - 188
Email: sales@nanotron.com
Internet: www.nanotron.com