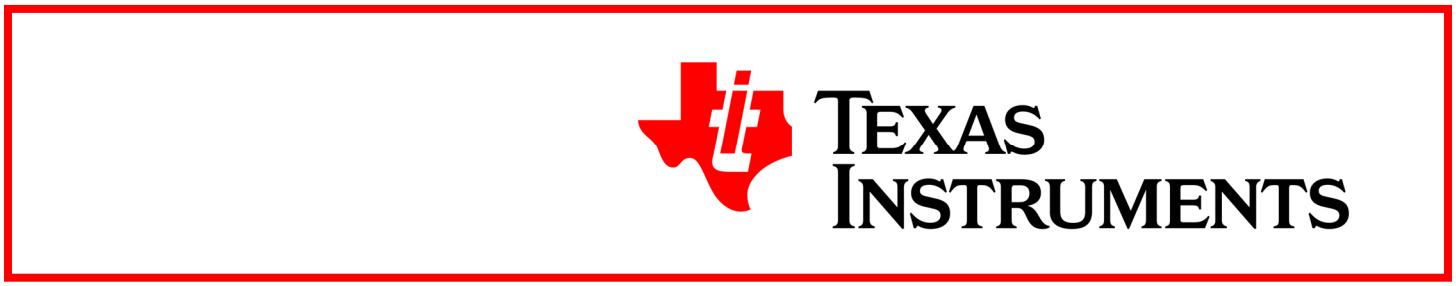
SimpliciTI



Application Programming Interface

Texas Instruments, Inc.

www.ti.com/simpliciti

Texas Instruments
Low-Power RF

Table of Contents

Table of Tables

i Revision History

Version	Description	Date
1.0	Initial release	08/01/2008
1.1	Update for the 1.1.0 release	01/14/2009
1.2	Updated title page	03/24/2009
1.3	Added updates for FHSS, updated formatting for consistency	10/29/2011

ii References

[1] SimpliciTI Specification Version 1.1.0

[2] SimpliciTI Developers Notes

iii Definitions

- API Application Programming Interface.
- BSP Board Support Package
- CCA Clear Channel Assessment
- GPIO General Purpose Input Output
- ISR Interrupt Service Routine
- LED Light Emitting Diode
- LQI Link Quality Indication.
- LRU Least Recently Used
- MAC Medium Access Control.
- PHY Physical layer.
- RSSI Received Signal Strength Indicator
- FHSS Frequency Hopping Spread Spectrum

iv Nomenclature

Fixed pitch fonts `such as this` refer to program source code or source file names.

1. Introduction

This document describes the application programming interface for SimpliciTI software. The API provides an interface to the services of the SimpliciTI protocol stack.

2. API Overview

2.1. Interface Mechanisms

The following interface mechanisms are used in the SimpliciTI API.

2.1.1. Direct Execute Function Calls

These API functions directly execute code that performs an operation. The function executes in the context of the caller. These functions may have critical sections.

2.1.2. Callback Function

There is one optional callback opportunity in SimpliciTI. The function must be defined and implemented by the application and is registered during initialization. The callback function implementation should avoid CPU intensive operations as it runs in the ISR context. This function is described in detail in Section 7.

2.1.3. Background Function

When compiling code with FHSS enabled, there is one function which must be called by user application code regularly in order to keep the network wide PLL operating properly. This is required because SimpliciTI is a single threaded system and thus background thread operation is partly the responsibility of the user application. Further explanation of this requirement can be found in the Application Note on Frequency Hopping document.

2.2. Data Interfaces

These interfaces support sending and receiving data between the SimpliciTI stack and the application and ultimately support the peer-to-peer messaging.

2.3. Common Constants and Structures

2.3.1. Common Data Types

The following are defined:

```
typedef signed char     int8_t;
typedef signed short    int16_t;
typedef signed long     int32_t;
typedef unsigned char   uint8_t;
typedef unsigned short  uint16_t;
typedef unsigned long   uint32_t;
typedef unsigned char   linkID_t;
typedef enum smplStatus smplStatus_t;
```

In addition a further set of types and structures are used for the ioctl interface. These are described in Section 6.

2.3.2. Status

The following status values are used in various API functions. They are of type **smplStatus_t**. The relevant return codes will be specified individually for each API symbol in the following sections.

NAME	DESCRIPTION
SMPL_SUCCESS	Operation successful.
SMPL_TIMEOUT	A synchronous invocation timed out.
SMPL_BAD_PARAM	Bad parameter value in call.
SMPL_NOMEM	No memory available. Object depends on API.
SMPL_NO_FRAME	No frame available in input frame queue.
SMPL_NO_LINK	No reply received to Link frame sent.
SMPL_NO_JOIN	No reply received to Join frame sent.
SMPL_NO_CHANNEL	Channel scan did not result in response on at least one channel.
SMPL_NO_PEER_UNLINK	Peer could not delete connection. Returned in reply message to unlink request. (Not officially supported yet.)
SMPL_TX_CCA_FAIL	Frame transmit failed because of CCA failure.
SMPL_NO_PAYLOAD	Frame received but with no application payload.
SMPL_NO_AP_ADDRESS	Should have previously gleaned an Access Point address but we have none.

Table 1 **smplStatus** result codes

2.3.3. Special Link IDs

SimpliciTI supports special Link IDs that are available to the application by default. The following values indicate the special Link IDs.

NAME	DESCRIPTION
SMPL_LINKID_USER_UUD	Unconnected User Datagram Link ID. This is a special, connectionless Link ID supported by default on all user applications.

Table 2 Special Link IDs

3. Initialization Interface

3.1. Introduction

SimpliciTI initialization involves three stages of initialization: board, radio, and stack. Board initialization (BSP) is deliberately separated from the radio and stack initialization. The radio and stack initialization occur as a result of the SimpliciTI initialization call. The board initialization is a separate invocation not considered part of the SimpliciTI API but it is noted here for completeness.

The BSP initialization is partitioned out because customers may already have a BSP for their target devices. Making the BSP initialization explicit in the SimpliciTI distribution makes it easier to port to another target.

3.1.1. Board Initialization

SimpliciTI supports a minimal board-specific BSP. The BSP scope includes GPIO pin configuration for LEDs, switches, and a counter/timer used for protocol chores. It also includes SPI initialization for the dual-chip RF solutions.

3.1.1.1. UART Driver

A UART driver is also included with SimpliciTI. If the driver is utilized by the user application it must also be initialized. Additional information can be found in the Application Note UART Driver document.

3.1.2. Radio Initialization

Radio registers are populated and the radio is placed in the powered, idle state. Most of the radio registers are based on exported code from SmartRF Studio. The default channel is set with the first entry in the channel table. For FHSS enabled systems, the default channel set includes 25 preset channels for the hop schedule. Note that for the 802.15.4 compatible radios the channel set provides a set of 5 channels in a pseudo random sequence which repeats every 25 hop events. For the non-802.15.4 compatible radios, the channel set is 25 separate channels in a pseudo random sequence.

3.1.3. Stack Initialization

All data structures and network applications are initialized. If Frequency Hopping is enabled, the network wide PLL is enabled and further initialization is delayed until the PLL is locked onto the Frequency Hopping hop schedule. In addition the stack issues a

Join request on behalf of the device. The Join request will fail in topologies in which there is no Access Point. This is expected in this topology and is not an error condition.

In topologies in which an Access Point is expected to be present Join failure is an error condition and the application should continue to retry or take other action.

3.2. `BSP_Init ( )`

3.2.1. Description

Not strictly part of the SimpliciTI API this call initializes the specific target hardware. It should be invoked before the **`SMPL_Init()`** call.

3.2.2. Prototype

```
void BSP_Init(void)
```

3.2.3. Parameter Details

None.

3.2.4. Return

None.

3.3. `SMPL_Init ( )`

3.3.1. Description

This function initializes the radio and the SimpliciTI protocol stack. It must be called at least once when the software system is started and before any other function in the SimpliciTI API is called. Note that is is acceptable to call this function multiple times as a pseudo Join request operation.

3.3.2. Prototype

```
smplStatus_t SMPL__Init(uint8_t (*callback)(linkID_t))
```

3.3.3. Parameter Details

PARAMETER	DESCRIPTION
`Callback`	Pointer to function that takes a **`linkID_t`** argument and returns a uint8_t.

A non-null argument causes the supplied function to be registered as the callback function for the device. Since the initialization is called only once the callback serves all logical End Devices on the platform.

The function is invoked in the frame-receive ISR thread so it runs in the interrupt context. Details of the callback are discussed in Section 7.

It is valid for this parameter to be null if no callback is supplied.

3.3.4. Return

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Initialization successful.
`SMPL_NO_JOIN`	No Join reply. Access Point possibly not yet up. Not an error if no Access Point in topology
`SMPL_NO_CHANNEL`	Only if Frequency Agility enabled. Channel scan failed. Access Point possibly not yet up.

4. Connection Interface

4.1. Introduction

This interface provides the mechanism to establish a connection between two peers.

4.2. `SMPL_Link()`

4.2.1. Description

This call sends a broadcast link frame and waits for a reply. Upon receiving a reply a connection is established between the two peers and a Link ID is assigned to be used by the application as a handle to the connection.

This call will wait for a reply but will return if it does not receive one within a timeout period so it is not a strictly blocking call. The amount of time it waits is scaled based on frame length and data rate and is automatically determined during initialization.

This call can be invoked multiple times to establish multiple logical connections. The peers may be on the same or different devices than previous connections.

4.2.2. Prototype

```
smplStatus_t SMPL_Link(linkID_t *lid)
```

4.2.3. Parameter Details

PARAMETER	DESCRIPTION
`Lid`	The parameter is a pointer to a Link ID. If the call succeeds the value pointed to will be valid. It is then to be used in subsequent APIs to refer to the specific peer.

4.2.4. Return

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Link successful.
SMPL_NO_LINK	No Link reply received during wait window.
SMPL_NOMEM	No room to allocate local Rx port, no more room in Connection Table, or no room in output frame queue.
SMPL_TX_CCA_FAIL	Could not send Link frame.

4.3. SMPL_LinkListen( )

4.3.1. Description

This call will listen for a broadcast Link frame. Upon receiving one it will send a reply directly to the sender.

This call is a modified blocking call. It will block “for a while” as described by the following constant set in the nwk_api.c source file:

CONSTANT	DESCRIPTION
LINKLISTEN_MILLISECONDS_2_WAIT	Number of milliseconds this thread should block to listen for a Link frame. The default is 5000 (5 seconds)

The application can implement a recovery strategy if the listen times out. This includes establishing another listen window. Note that there is a race condition in that if the listen call is invoked upon a timeout it is possible that a link frame arrives during the short time the listener is not listening.

4.3.2. Prototype

```
smplStatus_t SMPL_LinkListen(linkID_T *lid)
```

4.3.3. Parameter Details

PARAMETER	DESCRIPTION
Lid	The parameter is a pointer to a Link ID. If the call succeeds the value pointed to will be valid. It is then to be used in subsequent APIs to refer to the specific peer.

4.3.4. Return

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Link successful.
SMPL_TIMEOUT	No link frame received during listen interval. Link ID not valid.

5. Data Interface

5.1. Introduction

This API provides interfaces to send and receive data between peers.

5.2. SMPL_SendOpt ( )

5.2.1. Description

This function sends application data to a peer with the capability of specifying transmit options. The network code takes care of properly conditioning the radio for the transaction. Upon completion of this call the radio will be in the same state it was before the call was made. The application is under no obligation to condition the radio.

By default the transmit attempt always enforces CCA.

5.2.2. Prototype

```
smplStatus_t SMPL_SendOpt( linkID_t lid,
                           uint8_t *msg,
                           uint8_t len,
                           txOpt_t opts )
```

5.2.3. Parameter Details

PARAMETER	DESCRIPTION
`Lid`	Link ID of peer to which to send the message.
`Msg`	Pointer to message buffer.
`Len`	Length of message. This can be 0. It is legal to send a frame with no application payload.
`Opts`	Bit map of valid options selected for the transmit

The ‘lid’ parameter must be one established previously by a successful Link transaction. The exception is the Unconnected User Datagram Link ID (see Section 2.3.3). This Link ID is always valid. Since this Link ID is not connection-based a message using this Link ID is effectively a datagram sent to all applications.

Valid transmit options are:

Option (macro)	Description
SMPL_TXOPTION_NONE	No options selected.
SMPL_TXOPTION_ACKREQ	Request acknowledgement from peer. Synchronous call.

5.2.4. Return

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Transmission successful.
SMPL_BAD_PARAM	No valid Connection Table entry for Link ID; data in Connection Table entry bad; no message or message too long.
SMPL_NOMEM	No room in output frame queue.
SMPL_TX_CCA_FAIL	CCA failure. Message not sent.
SMPL_NO_ACK	No acknowledgment received.

5.3. SMPL_Send ( )

5.3.1. Description

This function sends application data to a peer. This API provides legacy support for SimpliciTI releases that predate the addition of the transmit options. This API is equivalent to calling **SMPL_SendOpt()** with **SMPL_TXOPTION_NONE** specified.

The network code takes care of properly conditioning the radio for the transaction. Upon completion of this call the radio will be in the same state it was before the call was made. The application is under no obligation to condition the radio.

By default the transmit attempt always enforces CCA.

5.3.2. Prototype

```
smplStatus_t SMPL_Send( linkID_t lid,
                        uint8_t *msg,
                        uint8_t len )
```

5.3.3. Parameter Details

PARAMETER	DESCRIPTION
`Lid`	Link ID of peer to which to send the message.
`Msg`	Pointer to message buffer.
`Len`	Length of message. This can be 0. It is legal to send a frame with no application payload.

The 'lid' parameter must be one established previously by a successful Link transaction. The exception is the Unconnected User Datagram Link ID (see Section 2.3.3). This Link ID is always valid. Since this Link ID is not connection-based a message using this Link ID is effectively a datagram sent to all applications.

5.3.4. Return

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Transmission successful.
`SMPL_BAD_PARAM`	No valid Connection Table entry for Link ID; data in Connection Table entry bad; no message or message too long.
`SMPL_NOMEM`	No room in output frame queue.
`SMPL_TX_CCA_FAIL`	CCA failure. Message not sent.

5.4. `SMPL_Receive ( )`

5.4.1. Description

This function checks the input frame queue for any frames received from a specific peer.

Unless the device is a polling device this call does not activate the radio or change the radio's state to receive. It only checks to see if a frame has already been received on the specified connection.

If the device is a polling device as specified in the device configuration file (see Section 9.2 in the Developers Notes) the network layer will take care of the radio state to enable the device to send the polling request and receive the reply. In this case conditioning the radio is not the responsibility of the application.

If more than one frame is available for the specified peer they are returned in first-in-first-out order. Thus it takes multiple calls to retrieve multiple frames.

5.4.2. Prototype

```
smplStatus_t SMPL_Receive( linkID_t lid,
                           uint8_t *msg,
                           uint8_t *len )
```

5.4.3. Parameter Details

PARAMETER	DESCRIPTION
Lid	Check for messages from the peer specified by this Link ID.
Msg	Pointer to message buffer to populate with received message.
Len	Pointer to location in which to save length of received message.

The ‘lid’ parameter must be one established previously by a successful Link transaction. The exception is the Unconnected User Datagram Link ID (see Section 2.3.3). This Link ID is always valid. The application must ensure that the message buffer is large enough to receive the message. To avoid a buffer overrun the best strategy is to supply a buffer that is as large as the maximum application payload specified in the network configuration file (MAX_APP_PAYLOAD) used during the project build.

5.4.4. Return

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Frame for the Link ID found. Contents of ’msg‘ and ’len ‘ are valid.
SMPL_BAD_PARAM	No valid Connection Table entry for Link ID; data in Connection Table entry bad.
SMPL_NO_FRAME	No frame available.
SMPL_NO_PAYLOAD	Frame received with no payload. Not necessarily an error and could be deduced by application because the returned length will be 0.
SMPL_TIMEOUT	Polling Device: No reply from Access Point.
SMPL_NO_AP_ADDRESS	Polling Device: Access Point address not known.
SMPL_TX_CCA_FAIL	Polling Device: Could not send data request to Access Point
SMPL_NOMEM	Polling Device: No memory in output frame queue
SMPL_NO_CHANNEL	Polling Device: Frequency Agility enabled and could not find channel.

6. Device Management: IOCTL Interface

6.1. Introduction

The `ioctl` interface is the means by which applications can get access to more refined control over the device. There is a general form for the interface that specifies an object, and action, and any parameters associated with the object and action.

The scope of this interface is large enough so that each form of control will be described in its own section below after the general interface format is described. Because the interface is so general it is easily extensible by customers.

6.2. Common constants and structures

The `ioctl` objects and actions are presented below. The parameter information supplied with the call varies widely depending on the object. The detailed parameter structure descriptions will be presented in the sections following the interface description when each individual interface is described.

6.2.1. `IOCTL` objects

The following objects are defined. Each will be discussed in a separate section following the general API description.

```
enum ioctlObject
{
  IOCTL_OBJ_FREQ,
  IOCTL_OBJ_CRYPTKEY,
  IOCTL_OBJ_RAW_IO,
  IOCTL_OBJ_RADIO,
  IOCTL_OBJ_AP_JOIN,
  IOCTL_OBJ_ADDR,
  IOCTL_OBJ_CONNOBJ,
  IOCTL_OBJ_FWVER,
  IOCTL_OBJ_PROTOVER,
  IOCTL_OBJ_NVOBJ,
  IOCTL_OBJ_TOKEN,
  IOCTL_OBJ_PLL
};
typedef enum ioctlObject   ioctlObject_t;
```

6.2.2. IOCTL actions

The following actions are defined. They will be discussed as they are relevant in the sections following the general API description.

```
enum ioctlAction
{
  IOCTL_ACT_SET,
  IOCTL_ACT_GET,
  IOCTL_ACT_READ,
  IOCTL_ACT_WRITE,
  IOCTL_ACT_RADIO_SLEEP,
  IOCTL_ACT_RADIO_AWAKE,
  IOCTL_ACT_RADIO_SIGINFO,
  IOCTL_ACT_RADIO_RSSI,
  IOCTL_ACT_RADIO_RXON,
  IOCTL_ACT_RADIO_RXIDLE,
  IOCTL_ACT_RADIO_SETPWR,
  IOCTL_ACT_ON,
  IOCTL_ACT_OFF,
  IOCTL_ACT_SCAN,
  IOCTL_ACT_DELETE
};
typedef enum ioctlAction   ioctlAction_t;
```

6.3. SMPL_Ioctl( )

6.3.1. Description

This is the single format taken by all ioctl calls.

6.3.2. Prototype

```
smplStatus_t SMPL_Ioctl( ioctlObject_t obj,
                         ioctlAction_t act,
                         void *val )
```

6.3.3. Parameter Details

PARAMETER	DESCRIPTION
`Obj`	Object of the action requested.
`Act`	Action requested for the specified object.
`Val`	Pointer to parameter information. May be input or output depending on action. May also be null if object/action combination requires no parametric information.

All instances of `val` in calls should be by reference, i.e., a true pointer. Do **not** cast the value of `val` to `void *`. The internal code dereferences the argument as if it were a pointer to the object. This can be inconvenient for a simple argument but has the advantage that the interface is completely consistent.

6.3.4. Return

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Operation successful.
`SMPL_BAD_PARAM`	ioctl object or ioctl action illegal.

Additional return values depend on object specified. These values will be described in the following sections.

6.4. IOCTL object/action interface descriptions

6.4.1. Raw I/O

6.4.1.1. Support structure definitions

The following structures support this object:

```
typedef struct
{
  uint8_t  addr[NET_ADDR_SIZE];
} addr_t;

typedef struct
{
  addr_t   *addr;
  uint8_t  *msg;
  uint8_t   len;
  uint8_t   port;
} ioctlRawSend_t;

typedef struct
{
  addr_t  *addr;
  uint8_t *msg;
  uint8_t  len;
  uint8_t  port;
  uint8_t  hopCount;
} ioctlRawReceive_t;
```

6.4.1.2. Interface details

This object permits sending to and receiving from arbitrary destination address/port combinations. Normally applications must have established peer connection using the linking scheme. This object permits unconditional communication. This support is used extensively by the **NWK** layer itself.

Note that this interface requires the caller to supply a complete Application address (device address and port number) not a Link ID as would be done from the application.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_RAW_IO	IOCTL_ACT_READ	ioctlRawReceive_t	**When executed returns the payload for the oldest frame on the specified port. It is similar to a** SMPL_Receive() **call except that additional information is available from the received frame.**
	IOCTL_ACT_WRITE	ioctlRawSend_t	**Sends the enclosed payload to the specified address/port combination.**

6.4.1.3. Return

6.4.1.3.1. IOCTL_ACT_WRITE

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Transmission successful.
SMPL_NOMEM	No room in output frame queue.
SMPL_TX_CCA_FAIL	CCA failure.

6.4.1.3.2. IOCTL_ACT_READ

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Frame for the Port found. Contents of 'msg' and 'len ' are valid.
SMPL_NO_FRAME	No frame available.

6.4.2. Radio Control

Some simple radio control features are currently available. At this time this interface does not support direct access to the radio configuration registers.

6.4.2.1. Support structure definitions

```
typedef int8_t rssi_t;

typedef struct
{
  rssi_t  rssi;
  uint8_t lqi;
} rxMetrics_t;

typedef struct
{
  linkID_t     lid;   /* input: port for which signal info
desired */
  rxMetrics_t  sigInfo;
} ioctlRadioSiginfo_t;

enum ioctlLevel
{
  IOCTL_LEVEL_0,
  IOCTL_LEVEL_1,
  IOCTL_LEVEL_2
};

typedef enum ioctlLevel ioctlLevel_t;
```

6.4.2.2. Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_RADIO	IOCTL_ACT_RADIO_SLEEP	NULL	Done before putting the MCU to sleep. Does a disciplined state change to the radio. Saves any radio registers necessary.
	IOCTL_ACT_RADIO_AWAKE	NULL	Done after MCU wakes up. Restores any radio registers necessary.
	IOCTL_ACT_RADIO_SIGINFO	ioctlRadioSiginfo_t	Get the signal strength information for the last frame received on the specified port.
	IOCTL_ACT_RADIO_RSSI	rssi_t	Get current RSSI value
	IOCTL_ACT_RADIO_RXON	NULL	Place radio in receive state
	IOCTL_ACT_RADIO_RXIDLE	NULL	Place radio in idle state to conserve power
	IOCTL_ACT_RADIO_SETPWR*	ioctlLevel_t	Set output power level.

* Enabled with EXTENDED_API build time macro definition.

6.4.2.3. Return

6.4.2.3.1. `Null` object

STATUS	DESCRIPTION
`SMPL_SUCCESS`	This call always succeeds.

6.4.2.3.2. `ioctlRadioSiginfo_t` object

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Receive metric information valid.
`SMPL_BAD_PARAM`	No valid connection information for Link ID specified in parameter structure.

6.4.2.3.3. `rssi_t` object

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	RSSI value valid. This call always succeeds.

6.4.2.3.4. `ioctlLevel_t` object

Status of request as follows:

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Specified power level valid and set.
`SMPL_BAD_PARAM`	Invalid power level specified.

6.4.3. Access Point Join Control

To add some control over the ability of a device to gain access to the SimpliciTI network the protocol uses tokens to both join a network and to create peers by linking. Additional control is provided by allowing an Access Point to exclude the processing of Join frames unless the context is set to permit such processing. The idea is that if the device cannot

join then it cannot obtain the proper link token for that network so it will not be able to link with any other devices.

6.4.3.1. Interface Details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_AP_JOIN	IOCTL_ACT_ON	NULL	Permit processing of Join frames.
	IOCTL_ACT_OFF	NULL	Ignore Join frames.

6.4.3.2. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.4. Device Address Control

This interface permits the application to override the build-time device address setting. If the application generates a device address at run time this interface is used to set that address. The setting of the address **must** occur before the call to `SMPL_Init()`. Otherwise the build-time address will prevail. Once the address is set under either condition (pre-initialization `ioctl` call or through `SMPL_Init()`) the address cannot be changed.

6.4.4.1. Supporting structure definition

```
typedef struct
{
  uint8_t  addr[NET_ADDR_SIZE];
} addr_t;
```

6.4.4.2. Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_ADDR	IOCTL_ACT_SET	addr_t	Sets address to value pointed to.
	IOCTL_ACT_GET	addr_t	Returns address in address pointed to.

6.4.4.3. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.5. Frequency Control

The current logical channel can be set and retrieved with this interface. A scan can also be requested. All of these interfaces are used by NWK in support of Frequency Agility.

Note: This interface is only supported if FREQUENCY_AGILITY is defined during compilation.

6.4.5.1. Supporting structure definitions

```
typedef struct
{
  uint8_t logicalChan;
} freqEntry_t;

typedef struct
{
  uint8_t      numChan;
  freqEntry_t *freq;
} ioctlScanChan_t;
```

6.4.5.2. Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_FREQ	IOCTL_ACT_SET	freqEntry_t	Sets logical channel to value pointed to.
	IOCTL_ACT_GET	freqEntry_t	Returns logical channel in address pointed to.
	IOCTL_ACT_SCAN	ioctlScanChan_t	Scans for replies on all logical channels. Channel numbers on which replies were received are returned in the freqEntry_t array pointed to.

6.4.5.3. Return

6.4.5.3.1. IOCTL_ACT_SET

STATUS	DESCRIPTION
SMPL_SUCCESS	Operation successful.
SMPL_BAD_PARAM	Requested logical channel number is out of range.

6.4.5.3.2. IOCTL_ACT_GET

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.5.3.3. IOCTL_ACT_SCAN

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds. However, the channel count in the returned parameter structure can be 0 which means that no channels were found. Caller should be sure to check the 'numChan' channel count element

6.4.6. Connection Control

Currently the following interface removes the connection entry for the specified Link ID. It does not tear down the connection by alerting the peer that the local connection is destroyed.

6.4.6.1. Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_CONNOBJ	IOCTL_ACT_DELETE	linkID_t	Deletes local connection from the connection table that is specified by the link ID pointer.

The Link ID SMPL_LINKID_USER_UUD is not a valid object for this call.

6.4.6.2. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	Operation successful.
SMPL_BAD_PARAM	Link ID is SMPL_LINKID_USER_UUD or no connection information for specified Link ID

6.4.7. Firmware Version

6.4.7.1. Supporting definitions

```
#define SMPL_FWVERSION_SIZE    4
```

6.4.7.2. Interface details

The firmware version that is running can be retrieved. It is a read-only (Get) object.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_FWVER	IOCTL_ACT_GET	uint8_t	Retrieves the current firmware version as a byte array.

The firmware version is an array of size **SMPL_FWVERSION_SIZE** that has the following format:

Byte	Contents
0	Major release number
1	Minor release number
2	Maintenance release number
3	Special release number

The values in each byte are binary.

6.4.7.3. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.8. Protocol Version

The protocol version can be used to determine interoperability context or to deny access. It is used during both the Join and Link negotiation. Currently the Join or Link is denied if the versions do not match. Backward compatibility could be implemented under some conditions.

6.4.8.1. Interface details

The current protocol version is read-only.

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_PROTOVER	IOCTL_ACT_GET	uint8_t	Protocol version.

6.4.8.2. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	This call always succeeds.

6.4.9. Non-volatile Memory Object

This object provides direct access to the current connection object. This object contains the context required to establish, maintain, and restore all peer connections. Other information is also kept such as store-and-forward client information if the device is an Access Point. An application can protect against reset conditions by saving and restoring this context appropriately.

The interface provides access to the object by providing an object version value, a length, and a pointer to the object. It is intended that the caller treat the object as a monolithic object and simply save or restore it as a single entity. The version and length information is supplied to help both with local handling and sanity checks when restoring the object.

This interface provides a GET action only. Application must do its own sanity checks. When saving a context the length and version elements in the ioctl object should be saved in addition to the monolithic NV object. When restoring a context the application should do a GET and then be sure that the object version and length elements match those that were previously saved.

This feature is enabled with EXTENDED_API build time macro definition.

6.4.9.1. Supporting structure definitions

```
typedef struct
{
  uint8_t    objVersion;
  uint16_t   objLen;
  uint8_t  **objPtr;
} ioctlNVObj_t;
```

6.4.9.2. Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_NVOBJ	IOCTL_ACT_GET	ioctlNVObj_t	Returns the version and length and a pointer to the connection context.

If the ‘objPtr’ element is null only the NV object version and length objects are populated.

Note that this interface provides (dangerous) direct access to the connection context area in memory. Care should be taken by applications to not disturb this memory or manipulate the contents directly.

6.4.9.3. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	
SMPL_BAD_PARAM	An action other than IOCTL_ACT_GET was specified.

6.4.10. Network Access Tokens

An interface is provided to get and set the two network access control tokens, the Join token and the Link token.

This feature is enabled with EXTENDED_API build time macro definition

6.4.10.1. Supporting definitions

```
enum tokenType
{
  TT_LINK,       /* Token Type is Link */
  TT_JOIN        /* Token Type is Join */
};

typedef enum tokenType tokenType_t;

/* If either token ever changes type a
   union will make things easier. */
typedef union
{
  uint32_t linkToken;
  uint32_t joinToken;
} token_t;

typedef struct
{
  tokenType_t  tokenType;
  token_t      token;
} ioctlToken_t;
```

6.4.10.2. Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_TOKEN	IOCTL_ACT_GET	ioctlToken_t	Get the value of the specified token into the 'token' object
	IOCTL_ACT_SET	ioctlToken_t	Set the value of the specified token from the 'token' object.

6.4.10.3. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	
SMPL_BAD_PARAM	A token other than TT_LINK or TT_JOIN or an action other than IOCTL_ACT_GET was specified.

6.4.11. Network Wide PLL (FHSS Enabled)

An interface is provided to turn on and off the network wide PLL object when FHSS is enabled. For those times when the system must sleep for extraordinarily long periods or control over radio traffic generated by the PLL subsystem is required, this allows the user to restart the network wide PLL once the system wakes up again. Starting the network wide PLL will cause the FHSS algorithm to seek and lock on to the most appropriate reference clock within radio range.

This feature is enabled with the FREQUENCY_HOPPING build time macro definition

6.4.11.1. Interface details

Object	Actions	(void *)val object	Comment
IOCTL_OBJ_PLL	IOCTL_ACT_ON	NULL	Turn on the network wide PLL and start the FHSS system
	IOCTL_ACT_OFF	NULL	Turn off the network wide PLL. FHSS is still active but the timing skew will grow linearly while the network wide PLL is off.

6.4.11.2. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	
SMPL_BAD_PARAM	A token other than IOCTL_OBJ_PLL or an action other than IOCTL_ACT_ON or IOCTL_ACT_OFF was specified.

7. Callback Interface

7.1. Introduction

A single callback may be registered during initialization by providing a function pointer as an argument to the initialization call (See Section 3.3). The function must be supplied by the application programmer.

7.2. Callback function details

7.2.1. Description

The callback (if registered) is invoked in the receive ISR thread when the frame received contains a valid application destination address.

7.2.2. Prototype

```
uint8_t sCallBack(linkID_t lid)
```

7.2.3. Parameter details

PARAMETER	DESCRIPTION
`Lid`	The Link ID of the connection bound to a received frame.

The parameter in the callback when invoked will be populated with the Link ID of the received frame. This is the way the callback can tell which peer has sent a frame and possibly requires service. The special Link ID SMPL_LINKID_USER_UUD is a valid parameter value in this context.

A call to SMPL_Receive() using the supplied Link ID is guaranteed to succeed1. This is the only means by which the frame can be retrieved.

7.2.4. Return

The callback must return either zero or non-zero. This value is the responsibility of the application programmer.

If the function returns zero the received frame is left in the input frame queue for later retrieval in the user thread. This is the recommended procedure. The callback can simply set a flag or otherwise store the information about the waiting frame. The actual call to **`SMPL_Receive()`** should be done in the user thread.

[1] The success is guaranteed unless the frame is deleted due to the LRU policy for managing the input frame queue. This can happen if the referenced frame is not retrieved in a timely manner.

If it returns non-zero the frame resource is released for reuse immediately. This implies that the callback has extracted all valid information it requires.

8. Extended API

8.1. Introduction

If the macro EXTENED_API is defined over the entire project build2 additional API symbols are enabled. These are described in the Sections that follow. The symbols are not enabled by default to save code space. If the macro is defined all the symbols are included.

8.2. `SMPL_Unlink( )`

8.2.1. Description

This API is used to tear down a connection in a disciplined manner. Disabling the connection consist of two actions. First, the local connection is unconditionally disabled. After this call any further references to the relevant Link ID will result in a return of SMPL_BAD_PARAM.

Second, a message is sent to the peer to inform the peer that the connection is being terminated. The calling thread will wait for a reply. If a reply is received it contains the result of the connection termination attempt on the peer. If a reply is not received the return from the call so indicates.

There is no guarantee that the message sent to the peer will be received. If the peer does not get the connection termination frame it must have some independent means to determine that the connection has been terminated

8.2.2. Prototype

```
smplStatus_t SMPL_Unlink(linkID_t lid)
```

8.2.3. Parameters

PARAMETER	DESCRIPTION
`Lid`	The Link ID of the connection to be disabled.

[2] This is done within the IAR IDE by defining the macro in the `smpl_nwk_config.dat` project file.

8.2.4. Return

Status of request as follows:

STATUS	DESCRIPTION
SMPL_SUCCESS	Unlink successful on both peers.
SMPL_BAD_PARAM	Link ID not found.
SMPL_TIMEOUT	No response from peer.
SMPL_NO_PEER_UNLINK	Peer did not have a Connection Table entry for specified connection.

8.3. SMPL_Ping( )

8.3.1. Description

This API implements the NWK Ping application on behalf of the User application. It pings the device associated with the peer specified. Note that it does not ping the peer itself but rather the device on which the peer is hosted. It is roughly equivalent to the ICMP application in the TCP/IP suite.

It is provided as a convenience for the User applications. It can be used to see if the hosting device is there. Since it does not talk to the peer itself it does not verify that the peer is there, but only that the device hosting the peer is there.

This API has the convenient side effect. If Frequency Agility is enabled it will scan the channels in the channel table if a reply is not received on the current channel. So, an application can discover a changed channel for free instead of implementing its own scan channel logic.

8.3.2. Prototype

```
smplStatus SMPL_Ping(linkID_t lid)
```

8.3.3. Parameters

PARAMETER	DESCRIPTION
Lid	The Link ID of the peer whose device should be pinged.

8.3.4. Return

STATUS	DESCRIPTION
SMPL_SUCCESS	Ping succeeded.
SMPL_TIMEOUT	No response from peer.

8.4. SMPL_Commission()

8.4.1. Description

This API is used to statically create a connection table entry. It requires detailed knowledge of the objects in the connection table. If both peers are (correctly) populated using this API a connection can be established without an explicit over-air linking transaction.

When used to create static connections User must know in advance the SimpliciTI address of the device for each peer. In addition, both local and remote port assignments must be made. The local port number on one device must correspond to the remote port assignment on the other device. They may have the same value but should be unique for each peer on a specific device.

The User port address space is partitioned into static and dynamic portions. The size of the static portion, the portion from which ports using this API must be drawn, is defined by the macro PORT_USER_STATIC_NUM found in the file .\Components\nwk.h. The default value is 1. The static port address space starts at 0x3E and builds down.

Range checks are made on the port assignments but other sanity checks, such as duplicate assignments, are not made.

8.4.2. Prototype

```
smplStatus_t SMPL_Commission( addr_t *peerAddr,
                             uint8_t locPort,
                             uint8_t rmtPort,
                             linkID_t *lid )
```

8.4.3. Parameters

PARAMETER	DESCRIPTION
peerAddr	Pointer to address of peer.
locPort	Local static port assignment
rmtPort	Remote static port assignment.
Lid	Pointer to Link ID object. Value assigned by NWK.

8.4.4. Return

STATUS	DESCRIPTION
`SMPL_SUCCESS`	Connection successfully created.
`SMPL_BAD_PARAM`	Bad Link ID pointer (value null) or ports out of range.
`SMPL_NOMEM`	No room in connection table.

9. Extended support

9.1. Introduction

In addition to the SimpliciTI API there are various support macros and functions available for use by applications. These are for convenience. As application examples were developed support in the form of certain “helper” utilities seemed sensible.

These are described in the following sections. The macros are defined in the file nwk_types.h.

9.2. NWK_DELAY( )

9.2.1. Description

This macro will implement a synchronous delay specified in milliseconds. It is not accurate so should not be used for time-sensitive applications.

It is used in the application examples for crude switch de-bouncing and delays between LED toggles to indicate application state.

9.2.2. Prototype (macro)

```
NWK_DELAY(uint16_t msDelay)
```

9.2.3. Parameter description

PARAMETER	DESCRIPTION
msDelay	Number of milliseconds to delay.

9.2.4. Return

N/A.

9.3. NWK_REPLY_DELAY( )

9.3.1. Description

An application can invoke this macro after sending a message to a peer from which it expects an immediate reply. The delay will terminate as soon as the next application frame is received (presumably the expected reply) or when a maximum time has expired. The maximum delay time is computed during initialization of the stack and is

scaled by the data rate and the maximum application payload size. It requires no user intervention.

A sample message exchange session using this macro is shown below.

9.3.2. Prototype (macro)

```
NWK_REPLY_DELAY()
```

9.3.3. Parameter description

N/A

9.3.4. Return

N/A

9.3.5. Example of macro usage

This code is incomplete in the sense that variable declarations and result code checks are not shown. However, the symbol references are all conformal.

```
/* Time to send a message to peer whose Link ID is 'linkID'
*/
{
  /* wake up radio */
  SMPL_Ioctl(IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_AWAKE, 0);

  /* Send message */
  SMPL_Send(linkID, &sendMsg, sizeof(sendMsg));

  /* Radio must be in Rx state to get reply. Then back to
   * IDLE to conserve power.
   */
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RXON, 0);
  NWK_REPLY_DELAY();
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_RXIDLE, 0);

  /* Get received reply */
  SMPL_Receive(linkID, &rcvMsg, &rcvMsgLen);

  /* radio off */
  SMPL_Ioctl( IOCTL_OBJ_RADIO, IOCTL_ACT_RADIO_SLEEP, 0);
}
```

9.4. `nwk_pllBackgrounder( )`

9.4.1. Description

An application must make regular calls to this function when FHSS is enabled. This function manages the network wide PLL and the FHSS hop schedule. The jitter of the FHSS hop schedule is inversely related to the rate at which this function is called, i.e. more frequent calls will result in less jitter and lower probability of communications failure.

Note that all calls to timing delay functions and also the macros NWK_DELAY and NWK_REPLY_DELAY automatically manage making calls to this function so the user application can freely insert significant timing delays using standard API's without needing to worry about managing the calls to this function.

Normally, calls to this function will return within a few tens of microseconds. However, when certain events occur, calls to this function can take additional time. There are three cases where this can happen.

- When the FHSS hop period expires, the radio must be turned off, the channel changed, and then turned on again. If calibration is enabled on the radio, this event can consume almost a full millisecond of time.

- When the network wide PLL determines it is necessary to request timing information from the associated reference clock radio, a PLL pump request on the nwk_pll port is generated. This operation can take a couple of milliseconds to complete. Note that the system does not wait for a reply to this request.
- When a reply to a PLL pump request is received, the function must make significant calculations to determine the actual error between the time measured on the current and reference radios. This operation can take about a millisecond but is entirely dependant on the speed of the processor.

The operation of the network wide PLL is intended to be capable of significant delays in pump requests including completely missing some of them. This allows for the no_pump parameter in the call. If this parameter is true, the events connected with PLL pump requests and responses are suppressed (events 2 and 3) and only the FHSS scheduling events (event 1) are allowed. This provides the user some additional control for tight timing situations to minimize the blocking nature managing associated events.

9.4.2. Prototype

```
bool nwk_pllBackgrounder( bool no_pump )
```

9.4.3. Parameter description

PARAMETER	DESCRIPTION
`no_pump`	If true, the FHSS hop schedule is maintained but no PLL pump radio requests are initiated. If false, only the FHSS hop schedule is maintained.

9.4.4. Return

Return value is true (non-zero) if the network wide PLL considers itself locked to the FHSS hop schedule, false otherwise. The user can utilize this information to determine if radio communications should be suppressed or not.