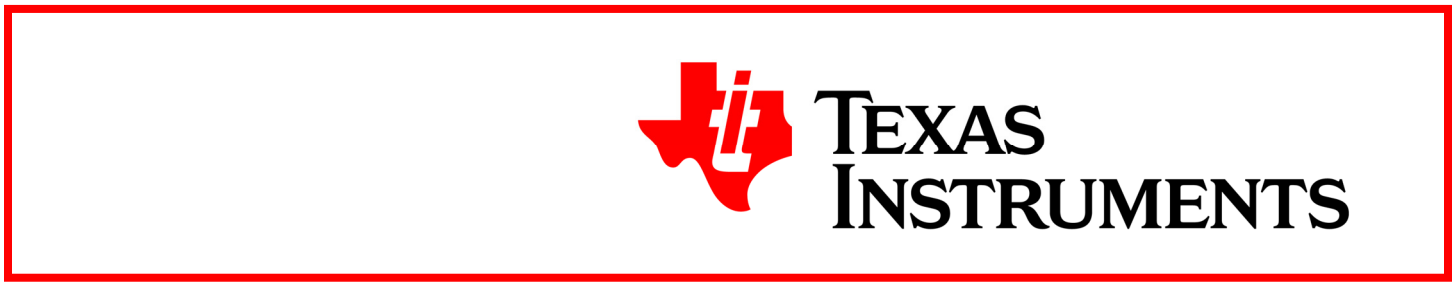
SimpliciTI



Users Guide
Frequency Hopping (FHSS)

Texas Instruments, Inc.

www.ti.com/simpliciti

Texas Instruments
Low-Power RF

Table of Contents

Table of Figures

i Revision History

Version	Description	Date
0.9	Development Specification	11/20/2010
1.0	Initial Public Release	11/07/2011

ii References

[1] SimpliciTI Specification Version

[2] SimpliciTI Developers Notes

[3] SimpliciTI Channel Table Information

[4] Application Note on SimpliciTI Frequency Agility Description

iii Definitions

- API Application Programming Interface.
- BSP Board Support Package
- CCA Clear Channel Assessment
- CCS Code Composer Sstudio (TMDFCCS-MCULTD)
- EW Embedded Workbench (toolset from IAR Systems)
- FHSS Frequency Hopping Spread Spectrum
- GPIO General Purpose Input Output
- IAR IAR Systems (http://www.iar.com/)
- IDE Integrated Development Environment
- 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
- SoC System on Chip (MCU and radio are on the same integrated circuit)
- SRF04 SmartRF04 Evaluation Board
- SRF05 SmartRF05 Evaluation Board (Rev 1.7 or later)

iv Nomenclature

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

1. Introduction

This user's guide describes the frequency hopping network application within the latest version of Texas Instruments' SimpliciTI radio communications stack.

Frequency hopping spread spectrum (FHSS) systems are used for many reasons some of which are robustness against narrow band noise and the ability to transmit at higher power as provided by the regulatory body in a given locale. Whatever the reason, the FHSS algorithm has traditionally been left to the application programmer for implementation which, until now, has unduly complicated coding efforts. With the SimpliciTI implementation of an FHSS system, the hopping algorithm is handled by background tasks freeing the application programmer to focus on other tasks.

This user's guide describes the FHSS system implemented as part of the SimpliciTI communications stack and the reasoning behind the particular implementation chosen.

Many aspects of this FHSS algorithm are capable of being controlled by the application programmer so that the FHSS algorithm may be optimized to operate in concert with a specific application. Tradeoffs are of course what must be considered when defining these elements of the code and this user's guide will attempt to instruct the user as to these issues as well.

Since SimpliciTI is not a multi-threaded library, there are some responsibilities the application programmer must adhere to in order for proper operation of the frequency hopping algorithm to work. That said, every attempt to minimize these dependencies on the application program have been made. These issues too, will be discussed within this user's guide.

Finally, the theory behind the chosen implementation of FHSS for SimpliciTI will be presented and discussed. This knowledge can be invaluable to system designers especially where system modeling is used in the design phase.

2. General Operation

FHSS algorithms, regardless of their form of implementation, ultimately require the set of radios comprising the network to coherently manage changing frequencies for communications on a given channel schedule. The difficulty underlying this task is the recognition that to coherently do anything requires a common time reference.

Traditionally FHSS algorithms have accomplished this via a beaconing approach where a master radio within the network broadcasts at regular intervals which the remaining radios (slaves) in the network use to synchronize their FHSS algorithms. The problem with this approach is that no two radios measure time in exactly the same way. Thus to make sure the slave radio has a high probability of receiving the beacon from the master, the slave radio must open a receive window of time which accounts for the possible error between the way the two radios measure time.

Unfortunately, if a slave radio misses the beacon for any reason, it must wait until the next beacon and open a receive window which is twice as large as the first window (assuming a constant beacon rate) in order to account for the increased possible error between the two radio's time measurements. If the slave radio misses the second

beacon, the receive window must now be three times as large… This process continues until a beacon is received.

The failure of this approach when implemented with an FHSS algorithm is that the timing skew measured at the moment the frequencies are changed increases linearly as well. If enough beacons are missed such that the receive window becomes as long as a single frequency dwell time under the FHSS algorithm, then the likelihood of failure approaches one hundred percent (by definition of the measurement error), thus making the system fragile. The only way to minimize this fragile condition is if the master beacons often enough that the measurement error will not create a receive window larger than a single frequency dwell time for the worst case beacon reception failure rate. Further, since the receive window must increase proportional to the sample period; the overhead of such a system is constant regardless of what the beaconing rate is. For battery operated systems, where the time interval between applications layer transmissions can be very long, this is problematic as the overhead can become the dominant consumer of power resources in the system.

The approach used within SimpliciTI implements frequency hopping through the use of a network wide PLL architecture thus providing an effective, coherent time source for all radios in the network. In this approach, each slave radio implements a software PLL mechanism which tunes its measurement of time to coincide with that of the master radio. Further, each radio dynamically adjusts the sampling period of its PLL so that it operates just above the noise floor allowing a constant receive window period regardless of how long between updates of the PLL are requested. Also, a novel differential measurement mechanism is implemented to further reduce timing skew between the master and slave radios at the moment frequency channels are changed thus increasing the overall success rate for application layer communications.

The PLL is managed by a NWK_PLL application layer which is almost completely transparent to the user application. The NWK_PLL application sends and receives packets in a background task that allow it to locate an acceptable reference clock and lock to its frequency and phase thus allowing coherent events (such as changes in frequency) to occur network wide.

3. Conversion of Existing Code

The general operation of SimpliciTI with frequency hopping enabled is very similar to that of using SimpliciTI operating on a single radio channel (it is suggested that users develop their application using a single frequency implementation and then enable frequency hopping during the final phase of the design process). The transition of code to systems in which frequency hopping is enabled should take minimal effort. This can be seen by the code snippets shown below. The first snippet shows the general form of a typical SimpliciTI main( ) function definition.

```
void main ( void )
{
  BSP_Init(); // initialize the board support package

  SMPL_Init(); // initialize SimpliciTI

  // application initialization code here

  while (1)
  {
    // application operational code here
  }
}
```

Note that only function calls are shown and not any parameters such as in the call to SMPL_Init( ). This is to emphasize the flow of operation and not the details of implementation.

In the following code snippet, the additional required support for frequency hopping is included.

```
void main ( void )
{
  BSP_Init(); // initialize the board support package

  SMPL_Init(); // initialize SimpliciTI

  // application initialization code

  while (1)
  {
    if( nwk_pllBackgrounder( false ) )
    {
      // application radio dependent operational code
    }

    // application non radio dependent operational code
  }
}
```

Notice that the only additional dynamic code required to support frequency hopping is a call to the function **`nwk_pllBackgrounder( )`**. This function handles all necessary operation of the frequency hopping algorithm within the constraints of the setup defined by the implementation. The return value from the **`nwk_pllBackgrounder( )`** function is a bool indicating whether or not it is considered safe to begin a transmission. The choice of whether or not it is safe is dependent upon the lock status of the nwk_pll application and also on the proximity of the next channel changing event as defined by the FHSS hopping schedule.

It is therefore suggested (but not required) that radio dependent code be wrapped in an "**`if`**" statement, as shown above, to test for the likelihood a given communication will

succeed. However, all application code that is independent of radio operations such as triggering an ADC sample or monitoring of external events can be implemented outside of this "`if`" statement. More detailed discussion of the **`nwk_pllBackgrounder( )`** function can be found in a subsequent section of this document.

The call to the **`SMPL_Init( )`** function normally issues a join request by default. Since frequency hopping systems must be changing frequencies coherently with other radios in the network, this call also initializes the frequency hopping code and waits until the dead band window is less than 5% of the frequency hopping rate before allowing the join request. Because a radio may be turned on at any point within the hopping schedule, the call to **`SMPL_Init( )`** may take quite some time to return depending on the hopping schedule, and the position within it, the network is at relative to the radio being turned on. This is required as the frequency hopping schedule must be locked onto by the radio before any join request can be expected to complete properly.

The system is very robust and once it has locked onto the frequency hopping schedule of the network, the application may use the radio as though it were communicating on a single frequency. The frequency hopping schedule is applied in the background (via the calls to **`nwk_pllBackgrounder( )`**) and respects the current status of ongoing radio communications. Thus, radio frequencies will not change during any active signaling between radios ensuring that as much radio communications as possible is successful for all radios regardless of the transmission source.

Additionally, for larger networks where one radio may not be able to directly communicate with the radio defined as the reference clock, other radios in the network can be defined as alternate reference clocks allowing them to reflect the reference clock radio's timer using their own timer as a surrogate reference clock. This allows for physically large networks such as might be encountered in networks for a campus of buildings or even whole municipalities.

4. Regulatory Compliance

The frequency hopping algorithm implemented within SimpliciTI is a general algorithm which should be capable of operating within any regulatory body's requirements. However, setting up the parameters to ensure regulatory compliance is the responsibility of the project implementer as it is beyond the scope of the code to manage all aspects of any regulatory body's requirements. Such things are usually, but not limited to, the same issues that a non-frequency hopping system would have to deal with. Out of band spurs and the like are examples of things beyond the scope of the code to manage and may require modifications to the operation of the radio on whole.

5. Managing Implementation Parameters

There are several compile time constants which the user can modify in order to tweak the frequency hopping algorithm and PLL operation to match a regulatory body's requirements and also to optimize operation between the frequency hopping code and the application being implemented.

All of these compile time constants are given a default value which can be overridden by the implementation in one of two suggested methods. Depending on the compiler used, these may not be the only possible ways of modifying these constants. Using one or

more of these methods should allow precise control over system operation without the need to change any of the library files thus eliminating the need to make local copies of the library for different projects.

5.1. Defining Constants at the Project Level

This method effectively uses the –D command line switch found in most compilers. If you are using an interactive development environment (IDE) as most compilers are now wrapped in, you generally access this option by adding token=value definitions in a project dialog box. In the IAR compiler this is accomplished via the pre-processor tab under the C/C++ category of the project options dialog as shown here.

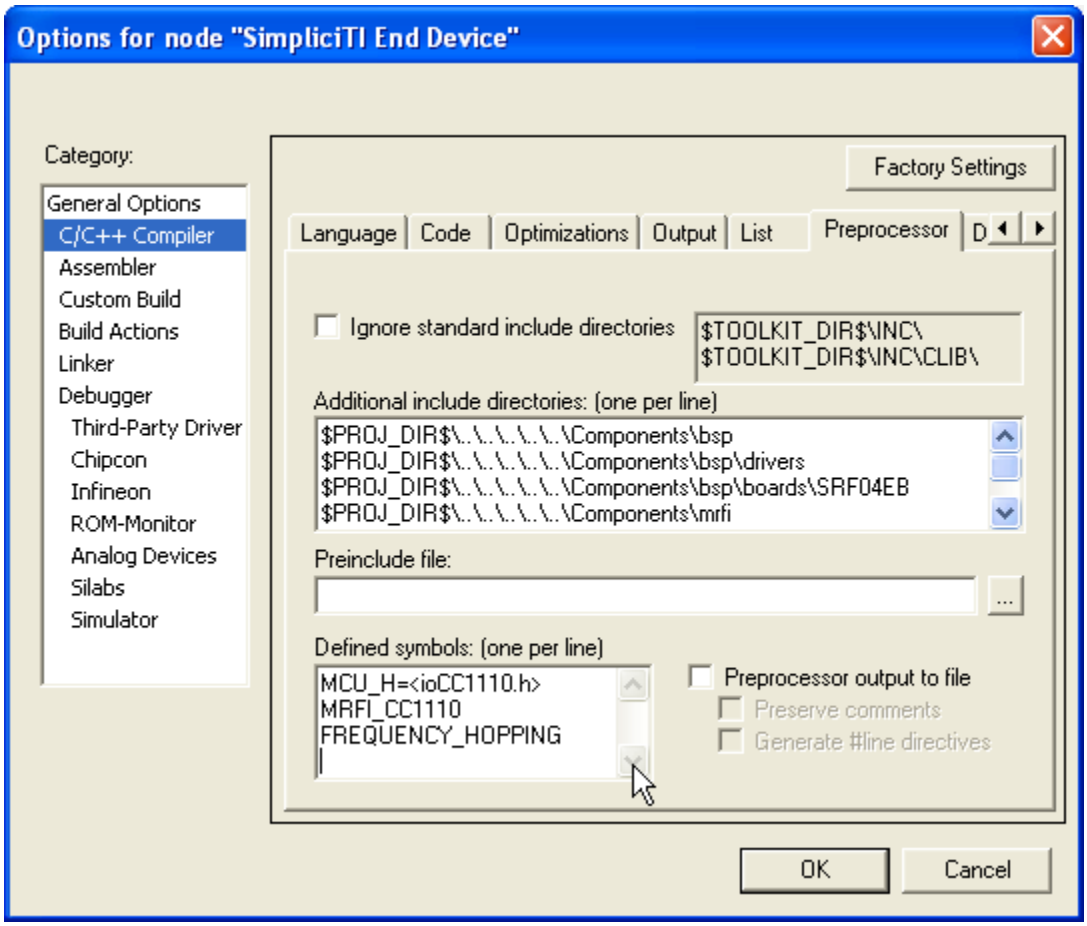


Figure 1 Enabling FREQUENCY_HOPPING with a macro

In this case, the token MCU_H is defined to be <ioCC1110.h> and the tokens MRFI_CC1110 and FREQUENCY_HOPPING are merely defined without any value.

5.2. Using a Project Level Include File

The second method may be easier to manage, especially if there are multiple subprojects within a solution. In this method, an extra header file for the entire project is

created with all of the #define constant values implemented as would be done normally using C syntax. Then this file is included in all translation units built for the project. In this manner, a separate command line switch is used which tells the compiler to include this header before any other file is read in a translation unit. Again, some IDE's provide access to this command line switch. In the IAR compiler it can be accessed in the pre-processor tab of the C/C++ category of the project options dialog as shown here.

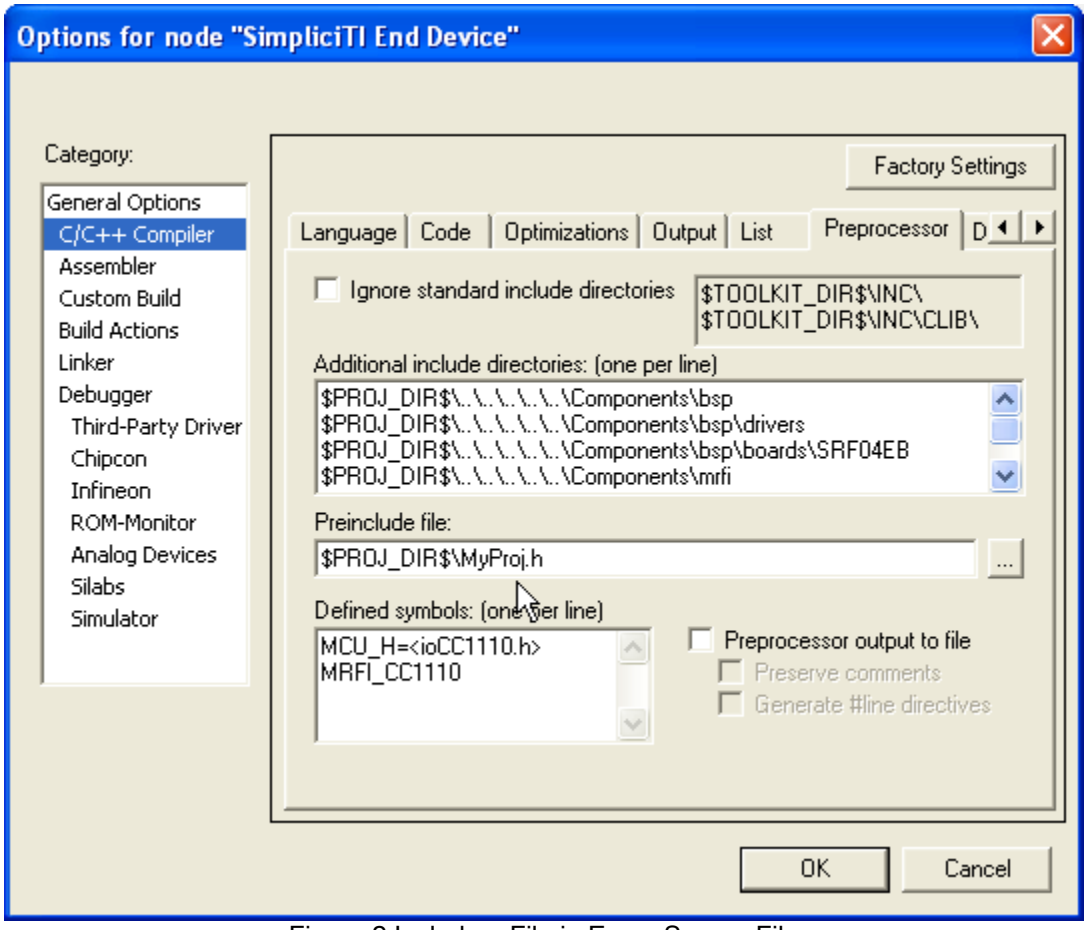


Figure 2 Include a File in Every Source File

In this case, the header file MyProj.h will be included in all files that are compiled. Presumably, MyProj.h will have definitions pertinent to the overall project implementation such as whether or not frequency hopping is enabled, etc.

6. Implementation Parameter Descriptions

Below are descriptions of those parameters which an implementation can modify to change the overall operation of the system with respect to the frequency hopping and PLL subsystems. All macros, except for the FREQUENCY_HOPPING and NWK_PLL_REFERENCE_CLOCK macros, provide a default value for use.

The following parameters are compile-time settings some of which must be the same for all radios on a given network; others can be unique to each radio depending on that

radio's application requirements. Those that must be consistent across the network are indicated in their description.

6.1. FREQUENCY_HOPPING

To enable frequency hopping this token must be defined in all translation units. Note that if the token FREQUENCY_AGILITY is also defined a compile error will be generated as these two algorithms are mutually exclusive in their operational methods. To correct the compile time error, remove the definition of the one you do not wish to implement.

All radios on the network must define this macro to operate properly in a SimpliciTI frequency hopping network.

Example:

```
#define FREQUENCY_HOPPING
```

6.2. BSP_TIMER_USED

This macro can be set to define which hardware timer is used to implement the PLL operation. For the 8051 base code it can be set to 1, 3, or 4; for the MSP430 based code it can be set to BSP_TIMER_A3 or BSP_TIMER_B7.

On the 8051 based radios, timer 1 is a 16 bit timer and is the recommended timer for operation under FHSS. Timers 3 and 4 are 8 bit timers and because of their limited ability to time a full 1ms, they increase processing overhead as multiple interrupts are required for every millisecond of time. This can eat into processing time available for application code. Further, due to the strict requirements of being able to respond to timer interrupts for proper operation, critical sections of code where interrupts are disabled must be scrutinized to make sure interrupts are not disabled for more than a single timer interrupt period. Using the 8 bit timers will require timer interrupts be serviced approximately every 200us at minimum.

Note: Regardless of which timer on which platform is chosen for use by the NWK_PLL application, its operational mode is set up the same way. That is it counts up to a threshold value in the compare zero channel and then is reset to zero. The user is free to use any additional channels as they wish, such as generating additional interrupts or toggling a GPIO signal but the operation of the timer itself must not be modified. Additionally, the threshold value of the timer is updated every 1ms by the PLL software and thus can change from the initial value programmed into it during initialization. Threshold values in other timer channels should not assume a constant value in the channel 0 threshold register.

This parameter can be different between radios allowing the choice of timer used to be optimized for the application on a given radio.

Example

```
#define BSP_TIMER_USED 1 // use timer 1 (the 16 bit timer)
```

6.3. MRFI_HOP_TIME_ms

This macro defines the time period between changes in frequency as implemented by the frequency hopping algorithm. The value is in milliseconds and defaults to 400ms if the implementation does not define it specifically.

This value can be as small as 20ms but due to processor bandwidth and the time required to change radio frequencies, it is suggested that this value be at least 50ms. Values less than 50ms will begin to have performance issues which are dependent upon many factors such as the number of radios in the network and processing time required by the application.

This parameter must be the same for all radios on the network.

Example:

```
#define MRFI_HOP_TIME_ms 300 // hop every 300 milliseconds
```

6.4. NWK_PLL_RX_TIME_STAMP_FIFO_SZ

This macro defines the number of receive time stamp structures that are available to the system for managing PLL packets. For most slave radios, only one is required but during the location operation two to four may provide a better choice for the reference clock radio. This is because when the PLL starts up, it broadcasts a locate packet in an attempt to find a reference clock to lock to.

In response to this broadcast locate packet, any viable reference clock will respond from which multiple packets will be received fairly quickly. The number of receive time stamps that can be allocated thus improves the probability of the slave radio to find “the” reference clock radio (if it is within range) or at least the closest alternative reference clock radio. However, the time stamp structures are not small (approximately 20 bytes each) and can take up considerable space in systems with limited memory. As always, it is up to the implementation to determine how these tradeoffs are managed.

The default value of this macro is 4.

Example

```
#define NWK_PLL_RX_TIME_STAMP_FIFO_SZ 4
```

6.5. NWK_PLL_SAMPLE_PERIOD_DEFAULT

This macro defines the default (minimum) number of milliseconds between requests the PLL makes to the reference clock radio. Requests by the slave radio will never happen faster than this value. Smaller values will allow the radio to lock onto the frequency hopping schedule quicker at the expense of increased initial overhead when the radio starts up. Larger values will reduce radio traffic overhead but can increase the time from startup to lock. The default value is 50 milliseconds which will generally allow a radio to lock to the network schedule in under a second (once the correct channel is located).

Note: This value effectively works to modify the gain of the PLL control loop and lower values result in increased loop gain. If the value gets too low, i.e. the gain gets too high, the loop could become unstable.

Normally, round trip times of PLL packets takes about 20 milliseconds. This is not to say that the application is locked out for this period of time, but rather about 20 milliseconds transpires from the transmission of a PLL request packet until the PLL response packet is received. This is partly due to the low priority given PLL packets allowing application code to access the radio in between and maintain as much throughput as possible. Although it is possible for the system to operate at the minimum value of 20 milliseconds, there is little performance boost achieved considering the overhead incurred from doing this.

This parameter can be different for different radios on the network as it only affects the operation of the PLL on the slave radio in which it is operating.

The default value of this macro is 50 milliseconds.

Example:

```
// set minimum sample period to 100 milliseconds
#define NWK_PLL_SAMPLE_PERIOD_DEFAULT 100
```

6.6. NWK_PLL_SAMPLE_PERIOD_MAX

This macro is used to set an upper limit on the PLL sample period. This is because the PLL algorithm attempts to increase the sample period as the system locks onto and closes in on the reference clock. Since crystal oscillators are highly stable, the system reduces the effort of maintaining lock in an inversely proportional manner to the relative error between the slave clock and the reference clock.

For a well behaved system, this time between samples can easily reach minutes to more than and hour depending on how long the radio has been associated with the network. For some systems this can be unacceptable as the knowledge that the slave has lost lock can only be determined when the PLL requests a sample from the reference clock. This macro is provided for those systems which require a minimum amount of checking for system lock, for example at least once per minute.

The default value for this macro is 231-1 milliseconds which is effectively infinity as the noise floor of the system should show itself long before this value is reached (not to mention it would take several thousand years for the algorithm to reach this value in the absence of noise).

This parameter can be different for different radios on the network as it only affects the operation of the PLL on the slave radio in which it is operating.

Example:

```
// set maximum sample period to 2 minutes
#define NWK_PLL_SAMPLE_PERIOD_MAX 120000L
```

Note: The L suffix must be used for numbers larger than 32767.

6.7. NWK_PLL_REFERENCE_CLOCK

This macro defines the radio within the system that is to be used as the reference clock. This macro must be defined for exactly one radio within a given network. The radio doesn't necessarily have to be any special radio and can even be an end device whether or not the network contains access points and/or range extenders. The only requirement is that one and only one radio's code be compiled with this macro defined[1].

The radio within a network that has this macro defined will be the network reference clock regardless of how accurately this radio's crystal oscillator counts time. All radio's within the network will lock, either directly or through alternate reference radios, to this radio's timer. Further, this radio will never send any request packets but only respond to request packets received from other radios on the network.

This macro has no value, it is merely defined or not defined.

Example:

```
// make this radio the reference clock for the network
#define NWK_PLL_REFERENCE_CLOCK
```

6.8. NWK_PLL_FAILURE_LIMIT

This macro defines the number of times for which PLL request packets are not replied to by a reference clock before the PLL resets itself and attempts to locate a new reference clock.

This value can be unique for each radio on the network.

Example

```
// double the default failure limit
#define NWK_PLL_FAILURE_LIMIT 20
```

6.9. NWK_PLL_LOOP_GAIN_Kp
NWK_PLL_LOOP_GAIN_Ki
NWK_PLL_LOOP_GAIN_Kd

These macros define the proportional, integral, and differential[2] gains of the PLL control loop compensator. These values should be in the range of -32767 to 32767 or the fixed point calculations could overflow and cause unpredictable results. For most systems the default values of these parameters should work well. They are provided as user definable inputs for those few implementations where tuning of the control loop provides a performance improvement over the default values.

Full understanding of the control theory required to properly tweak these gain values is beyond the scope of this manual. Additionally, the non-linear effects of modifying the sample rate must also be taken into account with any modifications of these values.

[1] This limitation is planed to be removed by providing dynamic reference clock determination by the network itself where any clock defined as a reference clock will default to an alternate reference clock if another reference clock with a lower radio address is discovered within the network.

[2] The differential gain default is currently zero. This macro may go away in future implementations of FHSS as most PLL's only require a PI control loop to operate properly.

These values can be unique to each radio on the network. For example, radios configured to operate as alternate reference clocks may need stiffer control loop settings as opposed to those radios that cannot communicate directly to the radio with the NWK_PLL_REFERENCE_CLOCK macro defined.

Example:

```
// set PLL compensator loop gains
#define NWK_PLL_LOOP_GAIN_Kp 50
#define NWK_PLL_LOOP_GAIN_Ki 3
#define NWK_PLL_LOOP_GAIN_Kd 25
```

7. Controlling the NWK_PLL application

The NWK_PLL application essentially runs without any user intervention required. However, the PLL operation can be controlled somewhat via calls to the SMPL_Ioctl( ) function. There are two operations available using this interface.

1. Turn on the PLL using the IOCTL_ACT_ON action
2. Turn off the PLL using the IOCTL_ACT_OFF action

Neither of these operations requires any data so the value pointer passed in the SMPL_Ioctl( ) function is NULL and the object type is IOCTL_OBJ_PLL. Thus the two possible calls would look like

```
SMPL_Ioctl( IOCTL_OBJ_PLL, IOCTL_ACT_ON, NULL );
SMPL_Ioctl( IOCTL_OBJ_PLL, IOCTL_ACT_OFF, NULL );
```

Turning on the PLL will cause it to begin the process of locating a viable reference clock within the network and, if one is found, attempt to lock to it.

Turning off the PLL will reset the PLL operation and halt any request packets being sent to the reference clock for timing information. However, the modulation value that was last used by the timer will remain in effect thus the PLL will coast at the last known modulation value when it was turned off. Frequency hopping will continue on the defined schedule except it may drift overtime relative to other radios on the network.

8. Using the `MRFI_Set_ms_Event( )` and `MRFI_Get_ms_Event( )` functions[3]

When frequency hopping is enabled, a timer is initialized to generate events at 1ms intervals. The NWK_PLL application layer hooks into this event via a call to MRFI_Set_ms_Event( ). It is allowable for an implementation to also hook into this event in order to take advantage of the existing timer operation for other uses.

A call to the MRFI_Get_ms_Event( ) will return the current event handler function that is installed. Similarly, a call to MRFI_Set_ms_Event( ) will set the pointer to the event handler function passed and return the event handler function that it is replacing. In either case, the implementation must ensure the call to any previously installed handler

[3] These functions are only available in SimpliciTI when frequency hopping is enabled.

is properly managed so as to not disrupt any other sections of code dependent upon calls to the 1ms event handler function.

The proper way to manage this is shown here

```
// a place for any existing event handler
MRFI_ms_event_t MyApp_existingEvent = NULL;

// a semaphore for enabling the 1ms event code
volatile bool MyApp_1msEvt_en = true;

// a semaphore for controlling the 1ms event code
volatile bool MyApp_1msEvt_sem = false;

// declare the event handler for the MyApp application
void MyApp_1msEvt( void );

// initialization code for the MyApp application
void MyApp_Init( )
{
  // some MyApp initialization code...

  // install MyApp's event handler
  MyApp_existingEvent = MRFI_Set_ms_Event( MyApp_1msEvt );

  // more MyApp initialization code...
}

// define the event handler for the MyApp application
void MyApp_1msEvt( void )
{
  // if there was a previous event handler
  if( MyApp_existingEvent != NULL )
    MyApp_existingEvent( ); // call it

  // assert the 1ms event semaphore if it is enabled
  if( MyApp_1msEvt_en != false )
    MyApp_1msEvt_sem = true;
}
```

Place the following code snippet in the applications main loop.

```
// check to see if the MyApp event code should run
if( MyApp_1msEvt_sem != false )
{
  // indicate event is being consumed
  MyApp_1msEvt_sem = false;

  // manage the 1ms event
}
```

Unfortunately, there is no viable mechanism to remove oneself from the call chain as this would require the knowledge of where in the call chain we have been displaced (if we have in fact been displaced). The above method uses an additional semaphore to enable and disable the event effectively acting as an on/off switch. In this fashion, the event code can be turned on or off while maintaining the call chain integrity of other installed events.

The key importance here is to not break the call chain so that all applications, including the NWK_PLL application are properly serviced by the interrupt. Also, it is important that processing in this event be kept to a minimum as it is done within the interrupt thread. The preferred method is to set a semaphore value (that is monitored by a non-interrupt thread) and return immediately.

9. Implementation Responsibilities and Issues

This section discusses overall implementation requirements and how they affect operations when the frequency hopping algorithm has been enabled. Following the guidelines set forth here will provide faster implementation results and less debugging in general.

9.1. Calling the `nwk_pllBackgrounder( )` function

Because the frequency hopping algorithm, by definition, is time dependent, the call to nwk_pllBackgrounder( ) should happen relatively often. This is because the actual change of radio frequency happens within this function rather than in an interrupt thread and the amount of jitter surrounding any change of frequency is directly proportional to the rate at which this function is called.

For example, if nwk_pllBackgrounder( ) is only called once every 10ms then the radio could be effectively blocked from talking to other radios for up to 10ms before the frequency is changed. If the frequency hopping algorithm hops every 100ms then as much as 10% of the time the radio will not be able to receive and thus possibly 10% of communications may be missed. It is important that the system designer understand this error is not reflected in the value returned by the function.

It should be noted the nwk_pllBackgrounder( ) function will return very quickly (a few microseconds usually) most of the time as PLL operations are intended to have a small amount of overhead in general. The overhead from this function call on radios defined as the reference clock or alternate reference clocks will generally increase linearly with the number of slave devices they service.

There is no limitation on the number of places within an implementations code where calls to the nwk_pllBackgrounder( ) function can be placed. Thus if the implementation has a function that takes considerable time to complete, it is suggested that additional calls to the nwk_pllBackgrounder( ) function be interspersed within this function to increase the overall average rate of calls to nwk_pllBackgrounder( ).

Additionally, the return value from nwk_pllBackgrounder( ) is only used to provide status to the implementation about the probability of successful radio communications. It need not be tested for if this status is not needed. For example, if a function takes a long time to complete but is not dependent upon any radio communications itself, then there is no need to monitor the status returned by calls to the nwk_pllBackgrounder( ) function.

9.2. Atomic Processing

In any system where semaphores are used to communicate between separate threads of processing (like between interrupt code and background code as in SimpliciTI), atomic operations must be performed and this is usually accomplished by disabling interrupts. In SimpliciTI, this is accomplished via the function like macros BSP_CRITICAL_STATEMENT( ), BSP_ENTER_CRITICAL_SECTION( ), and BSP_EXIT_CRITICAL_SECTION( ) macros.

Since the timer interrupt is used to manage the timer modulation as well as the one millisecond counter, it is imperative that these interrupts not be blocked to the extent that some are missed. This puts the burden on the implementation to ensure that use of the critical section macros and any other forms of blocking interrupt processing are done in a very judicious manner. Most importantly, the implementation should take into consideration the time required of any function calls made while interrupts are disabled.

In many cases, long sections of code in which interrupts are disabled, can be broken into smaller subsections where interrupts can be enabled and then disabled immediately again, thus allowing for an interrupt to be processed. Alternately, if only one interrupt thread needs to be blocked because others do not affect the atomic processing to be done, then block only it and allow other interrupts to run normally (this requires user defined operations as the BSP_xxx macros work only on a global scope).

In all frequency hopping implementations, careful attention of all atomic operations and its impact on interrupt processing can significantly increase the likelihood of proper operation and minimize difficult to locate bugs in code.