MICROSOFT RESEARCH

Brick Specification

The Microsoft Research Software Radio (Sora) Project

The Sora Core Team
(Ver 1.6)
Dec, 2011

The Microsoft Research Software Radio (Sora) Project is an initiative from the Wireless and Network Group, Microsoft Research Asia.

Contents

Chapter 1.
Introduction

Software radio increasingly becomes a powerful tool for academic wireless research as well as industry prototyping due to its flexibility and programmability. Implementing wireless protocols on software radio in practice, however, remains a challenging task. It is mainly due to the lack of an efficient modular software architecture for high-performance digital signal processing (DSP). This document describes Brick, a flexible, modular software architecture for creating baseband processing programs on Sora software radio platform. The baseband program is constructored from fine-grained building blocks, called *bricks*. These brick components are interconnected to form a directed graph. The connection between two bricks contains a typed queue, where data are written to the queue by the upstream brick and read by the downstream brick. A brick has very limited basic interfaces, consisting *reset, process* and *flush*. For future extension, the user may write new bricks, new interface functions, or compose different processing graph with existing bricks.

The Brick programming model is basically designed for *synchronized* data flow processing. In such model, each component is data driven. A source brick may pump data into the processing graph. Each brick in the graph will read fixed number of data from its input buffer, process on them and produce a constant number of output data. The processing results are eventually collected by several sink bricks. Bricks in the graph share a storage called *context*, where bricks may read configuration parameters or store shared states. This context enables flexible interaction among bricks and makes the configuration of each individual brick easier. Finally, a big processing graph may be easily partitioned into several sub-graphs, which are pipelined among different CPU cores to speed the whole data processing.

The Brick is implemented as a C++ template library on general-purpose PC hardware and is a part of Microsoft Research Software Radio (Sora) platform. In the rest of the document, we describe the Brick programming model in detail, including the library implementation and its usage.

Chapter 2.
Brick Programming Model

A *brick* represents a basic processing component of a synchronized data flow. A brick respresents a conceptually simple operation, such as scrambling, symbol maping, resampling or Fast Fourier Transform. A complex processing, for example, baseband decoding, may contain several bricks that are interconnected into a directed graph. The connection between two bricks is a typed queue, called *Pin Queue*. A brick starts to process when there are enough data in its input pin queue and the processing results are written to the output pin queue. A brick will explicitly call the downstream brick's *process* interface when there are data having been put into input pin queue of the downstream brick.

There are five types of bricks:

- Source. A source brick contains only single output port (Figure 1(a)). Typically, a source reads raw digital samples from memory or a device, and pushes the data through its output pin queue to the downstream brick. There is usually one single source for each processing graph.
- Sink. A sink brick contains only one input port but no output port (Figure 1(b)). A sink locates at the end of a processing graph and collects the processing results from its input port. The result data may further store into memory or a device. A processing graph may contain multiple sinks.
- Filter. Filter bricks are most common components in the processing graph (Figure 1(c)). A filter contains one input port and one output port. A filter brick reads data from its input pin queue, processes them and writes the results to the output pin queue.
- Demux. A demux[1] brick contains one input port and multiple output ports (Figure 1(d)). A demux brick reads the input port and selectively (usually based on its internal state) to write results to one or several output ports.

[1] Demultiplexer.

- Mux. A mux[2] brick contains multiple input ports but only one output ports (Figure 1(e)). In current specficiation, all input ports of a mux brick must have the same data type. As a consequence, it may be simply regarded as a filter brick with multiple upstream bricks and it shares the same implementation of a filter brick.

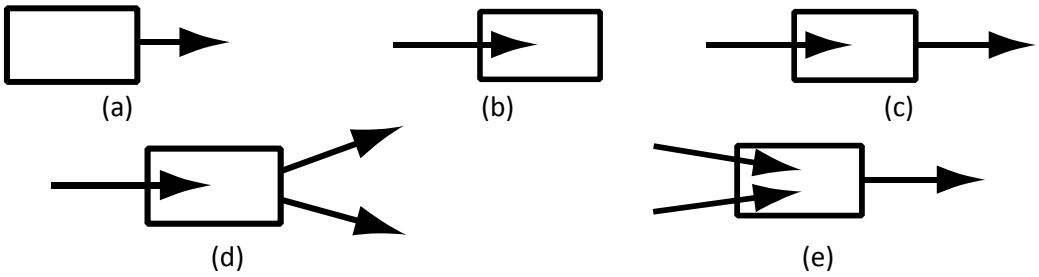

Figure 1. Brick types. (a) Source; (b) Sink; (c) Filter; (d) Demux; (e) Mux.

Figure 2 shows a sample graph for 802.11 modulation process. The PHY11b source obtains a MAC protocol data unit (MPDU) – a MAC frame for transmission, adds the preamble - synchronization bits and the physical layer convergence procedure (PLCP) header, then passes data byte-by-byte down through the graph. The 11b Multi-rate (MR) Selector is a demux brick that dynamically chooses DBPSK or DQPSK mapper based on the configuration and the data type. The Barker spreader is a filter brick, but acting as a mux. It may receive mapped symbols from multiple upstream brick and spreads each symbol to a high-frequency chip sequence. The chip sequence is then passed to the pulse shaper to generate the I/Q samples ready for transmission. The resulted I/Q samples are collected by the modulation sink in a memory buffer.

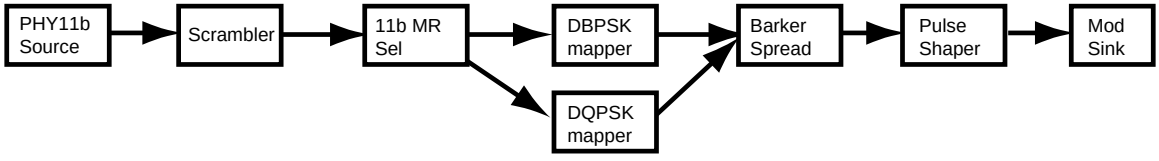

Figure 2. A sample graph for 802.11 modulation process.

[2] Multiplexer.

2.1 Brick Ports

A brick may have one input port (except for a source) and one or multiple output ports (except for a sink). A downstream brick always connects its input port to one outport of an upstream brick through a pin queue data structure. A port is characterized by a data type and a burst length for each read (input) or write (output). Macro **DEFINE_IPORT** and **DEFINE_OPORT** are used to declare the inport and outpout type of a brick. Figure 3 shows a sample inport and outport declaration of the Scrambler brick. The Scrambler will read one unsigned char from its input port and after scrambling, it outputs another byte to the outport.

```
// input
DEFINE_IPORT (uchar, 1);
// output
DEFINE_OPORT (uchar, 1);
```

Figure 3. Sample declarations of Brick ports.

To connect two bricks, one should ensure the data types of the inport and the outport match each other, though the burst length can be different. A compile-time error will occur if a user tries to connect two ports with different data type. Macro **DEFINE_OPORTS** can be used to declare the port type for each outport for a demux brick.

A brick only allocates buffers (called pin queue) for it output ports. When a brick schedules its downstream brick, it explicitly passes the pin queue as a parameter of the *process* method of the downstream bricks.

2.2 Read and Write Pin Queues

Pin queue is designed for synchronized data flow applications. Each read from (write to) a pin queue will obtain (output) a constant number of typed data. Before read a pin queue, a brick should first call **check_read** method. The method returns *true* if the pin queue has enough data to read. To access the data, the brick should call **peek** method, which will return a pointer referring to the memory block that holds the data. The **pop** method removes the data from the pin queue. Before write to a pin queue, the brick calls **append** method to allocate a memory buffer at the tail of the output pin queue. After filling the memory buffer, the brick should

explicitly call the downstream brick's *process* function, where the newly writtendata are read and continuously being processed.

2.3 Brick Interfaces

A brick defines three basic interface methods:

- **Reset**. The **Reset** method is used to reinitialize the brick. Usually, it causes the brick to read configuration parameters from the *context*, and assign internal states to initial values.
- **Process**. The **Process** method takes a pin queue as its parameter. This is the main method that a brick implements the data processing. The **Process** method returns a Boolean value. If the return value is **true**, the brick has processed the input data successfully and the brick is ready to receive further data. Otherwise the return value is **false**, it means the brick expects the data flow should be ended - e.g., the brick may catch an exception during the processing. This return value provides a convienent channel for a brick to signal the upstream about an exception. Note that this signaling is advisory, the upstream bricks may ignore it.
- **Flush.** The **Flush** method ends a data flow. When the **flush** method is called, the brick should clear its internal buffer and output all remaining data, if any, to the downstream.

2.4 Facade, Local and Graph Context

A brick is basically a C++ template class. It can has its internal states. However, there are several situations that requires a brick to expose its states. Firstly, the brick may need configuration paramters. Each time the brick is reseted, it should reload these parameters. Second, several bricks may share same set of states. For example, a multi-rate demod selector may needs the modulation type of the symbol stream to choose corrent demodulation path. But this modulation type information is obtained at a PLCP header parser brick. The parser brick needs to share this information with the selector brick.

A *facade* is a collection of shared states that a brick wants to expose. A facade is defined as a C++ structure type usually prefixed with **CF_**. Figure 4 shows a simple facade definition. Macro

FACADE_FIELD is used to define a data member as well as the access function that returns the reference to the data.

```
class CF_ScramblerSeed
{
    FACADE_FIELD (UCHAR, sc_seed);
};
```

Figure 4. A facade exposed by the scrambler brick. It contains a single state holding the default scrambler seed.

The union of all facades that a brick refers to, including all it exposes, is called *Local Context*. Macro **DEFINE_LOCAL_CONTEXT** is used to define the local context of a brick. In the brick definition, it should also explicitly refer to the local context using macro **REFERENCE_LOCAL_CONTEXT.** Figure 5 shows a sample definition of the local context. If one brick exposes multiple facades, they should all be passed as parameters to the macro **DEFINE_LOCAL_CONTEXT.** Specially if one brick exposes none shared states, it should use CF_VOID as a placehold facade parameter for the macro

```
DEFINE_LOCAL_CONTEXT.DEFINE_LOCAL_CONTEXT(TSc741, CF_ScramblerSeed);
```

Figure 5. The definition of the local context of TSc741 brick. The brick refers to single facade.

The *Graph Context* is the union of all bricks' local contexts in the processing graph. Before building a processing graph, an graph context object should be created. This graph context object will be passed to each brick's constructor when the processing graph is constructed. Figure 6 shows a sample graph context definiation of an 802.11b modulator.

```
struct FB11bModContext :
        LOCAL_CONTEXT(TModSink)
      , LOCAL_CONTEXT(TQuickPulseShaper)
      , LOCAL_CONTEXT(TBB11bDBPSKSpread )
      , LOCAL_CONTEXT(TBB11bDQPSKSpread )
      , LOCAL_CONTEXT(TBB11bMRSelect)
      , LOCAL_CONTEXT(TSc741)
      , LOCAL_CONTEXT(TB11bSrc)
{
};
```

Figure 6. A sample graph context definiation of an 802.11b modulator.

2.5 Reference to variables in Context

A brick uses macro **BIND_CONTEXT** to bind a variable in the graph context to a reference defined in the brick. After binding, the brick can use the reference to access the context variable. Compared with directly access the variable in the graph context, it optimize the memory access cost introduced by context implementation (virtual inheritance).

The reference in the brick is defined by the macros CTX_VAR_RO or CTX_VAR_RW. CTX_VAR_RO means the reference can be read only, ie. reference to constant. CTX_VAR_RW means the reference can be read or written, ie. reference to normal variable.

Figure 7 shows a sample for using **BIND_CONTEXT**. **m_seed** is a read-only reference to an unsigned char variable. The **BIND_CONTEXT** macro in the figure binds **m_seed** to **CF_ScramblerSeed::sc_seed** in the graph context.

```
CTX_VAR_RO(UCHAR, m_seed);
…

STD_TFILTER_CONSTRUCTOR(TSc741)
   BIND_CONTEXT(CF_ScramblerSeed::sc_seed, m_seed)
{
   …
}
```

Figure 7. A sample of binding context variable.

2.6 Event

When declaring a facade, it is also possible to define an event handler. For example, a brick performing clear-channel assess (CCA) may raise an event called *power_detected*. Therefore, in facade CF_11CCA, an event handler **OnPowerDetected** may be defined as follows,

```
FINL void OnPowerDetected() { }
```

The default behavior is just do nothing. But it can be overloaded in the graph context. When such an event is detected (i.e., the channel is in high power state), the CCA brick may invoke the event handler using **RaiseEvent** macro. For example, in TEnergyDetect brick, the following code will invoke the **OnPowerDetected** handler,

```
if ( average_energy_ >= cca_pwr_threshold ) {
    cca_state = CF_11CCA::power_detected;
    RaiseEvent(OnPowerDetected)();
}
```

2.7 Parameterized Brick

A brick may be parameterized, meaning that you can provide values to specialize it when creating the brick instance. This is achieved through nested template classes. For example, an FFT brick may take a parameter *N* to specify the number of points on which FFT is performed. The following Figure 8 shows the code snippet defines the parameterized brick TFFT. Then, **TFFT<64>::Filter** specializes an FFT filter that performs 64-point FFT.

```
DEFINE_LOCAL_CONTEXT(TFFT, CF_VOID);
template<int N> TFFT {
template<TFILTER_ARGS>
class Filter: public TFilter<TFILTER_PARAMS>
{
public:
   DEFINE_IPORT  (COMPLEX16, N);
   DEFINE_OPORT(COMPLEX16, N);

public:
   REFERENCE_LOCAL_CONTEXT(TFFT);
   STD_TFILTER_CONSTRUCTOR(Filter)   {}

   BOOL_FUNC_PROCESS(ipin)
   {
      while (ipin.check_read())
      {
         vcs *input   = reinterpret_case<vcs*>(ipin.peek());
         vcs *output = reinterpret_case<vcs*>(opin0().append());

         FFT<N>(input, output);

         ipin.pop();
         Next()->Process(opin0());
      }
      return true;
   }
};
};
```

Figure 8. Parameterizing a brick with nested template classes.

2.8Define Brick Processing Graph

Brick provides convienent macros to help define a processing graph. The declaration of a processing graph always starts from sinks, and then the upstream bricks until the source is reached. In current version, Brick supports only a single source per processing graph. Normally, a SDR application may have multiple processing graphs. For example, a normal transceiver

should at least have two processing graphs: one for the modulation process and another for the demodulation process.

Macro **CREATE_BRICK_SINK** is used to create a sink, and macro **CREATE_BRICK_FILTER** is used to create a filter brick. Macro **CREATE_BRICK_DEMUX##** declares a demux brick. The **##** mark represents a number between 2 to 10, which specifies the number of outports of the demux brick. Figure 9 shows the syntax of the above **CREATE_BRICK_XXX** macros. Parameter **name** specifies the brick object name, **type** specifies the brick type, **context** is the graph context object, and **next** or **next#** specifies the downstream brick connected to the brick's #th outport.

```
CREATE_BRICK_SINK          ( name, type, context );
CREATE_BRICK_FILTER        ( name, type, context, next);
CREATE_BRICK_SOURCE        ( name, type, context, next );
CREATE_BRICK_DEMUX##   (name, type, context, next0, …);
```

Figure 9. The syntax of the CREATE_BRICK_XXX Macros.

Figure 10 shows a code snippet creates an 802.11b modulater. The function returns an **ISource** pointer. **ISource** defines an interface to invoke the processing graph. Figure 11 shows the definition of the **ISource** interface. A thread may call ***ISource::Process*** repeatly for data processing until the function returns false, or other events occur indicate the process should be ended.

```
FB11bModContext BB11bModCtx;

ISource* CreateModGraph () {
  CREATE_BRICK_SINK     ( drop,    TDropAny,  BB11bModCtx );

  CREATE_BRICK_SINK   ( modsink,       TModSink, BB11bModCtx );
  CREATE_BRICK_FILTER ( pack16to8, TPackSample16to8,BB11bModCtx, modsink );
  CREATE_BRICK_FILTER ( shaper,       TQuickPulseShaper, BB11bModCtx, pack16to8);


  CREATE_BRICK_FILTER ( bpskSpread, TBB11bDBPSKSpread, BB11bModCtx,shaper);
  CREATE_BRICK_FILTER ( qpskSpread, TBB11bDQPSKSpread, BB11bModCtx,shaper);

  CREATE_BRICK_DEMUX2 ( mrsel, TBB11bMRSelect, BB11bModCtx,
       bpskSpread, qpskSpread );
  CREATE_BRICK_FILTER ( sc741,  TSc741, BB11bModCtx, mrsel );


  CREATE_BRICK_SOURCE ( ssrc, TBB11bSrc, BB11bModCtx, sc741 );
  return ssrc;
}
```

Figure 10. The sample code creates an 802.11b modulator. The function returns an ISource point.

```
struct ISource
{
    virtual bool Process () = 0;
    virtual void Reset ()   = 0;
   virtual void  Flush ()   = 0;
};
```

Figure 11. ISource interface.

2.9 Inline Optimization

The brick library utilizes the inline optimization of C++ to speed up the processing. However, extensively use inline function expansion may greatly enlarge the code size, especially in a complex progressing graph. An overly large code base may cause instruction cache misses (e.g., long jumps to a code piece that is outside the L1 cache) and may eventually decrease the processing speed. The default behavior of the brick library forces inline function expansion. But the programmer can overwrite this behavior to prevent the code from growing too large by adding a **TNoInline** filter between two bricks. TNoInline performs nothing (and thus adding no runtime overhead) but stops the inline function expansion of the downstream brick's process method during compilation.

2.10 Multi-Threading

A Brick graph may be seperated into several sub-graphs, each of which may run in a Sora thread. The predefined brick type **TThreadSeparator** is used to split the processing graph into multiple components. TThreadSeparator works like a filter brick in the processing graph and is created by the normal macro **CREATE_BRICK_FILTER**, but it exposes an **ISource** interface that can be called by a separate thread. Figure 12 illustrates an example that uses TThreadSeparator to partition a brick processing graph into two different threads. One thread (thread1) will periodically call the ISource interface of Source A1 and the second thread (thread2) will invoke the ISource interface exposed by TThreadSeparator.

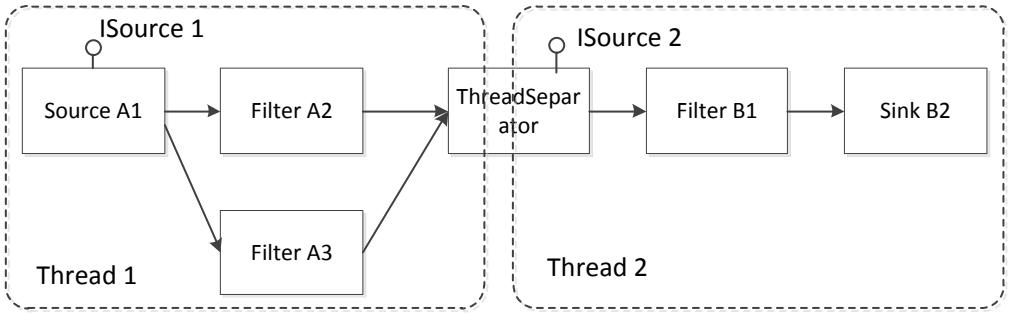

Figure 12. Using TThreadSeparator to partition a graph into two threads.

The internals of TThreadSeperator are illustrated in Figure 13. The left-side of the dash line is located in one thread and the right-side logics are running in another thread. A circular buffer isolates the data path between the two threads. When the Process method of TThreadSeparator is called by the upstream brick, it will simply put the data into the circular buffer. If there are no available slot in the buffer, the Process method is blocked. The second thread will periodicaly call TThreadSeparator's ISource::Process method. This method will check the circular buffer, copy the available data into pinqueue that connects the downstream brick, and schedules the process of its downstream brick (indicated by *Next* in the figure). Reset and Flush events are processed in the similar way. When the Reset (or Flush) method is called, an event variable, evReset(evFlush) is set. Then, in the second thread, the ISource::Process method will check the variable state and propagate the event downstream.

IMPORTANT NOTICE: In current implementation (Sora version 1.6), ISource interfaces, both exposed by Source bricks or TThreadSeparators, MUST NOT share the same thread. Otherwise, deadlocks may occur. This limitation will be removed in later Sora releases.

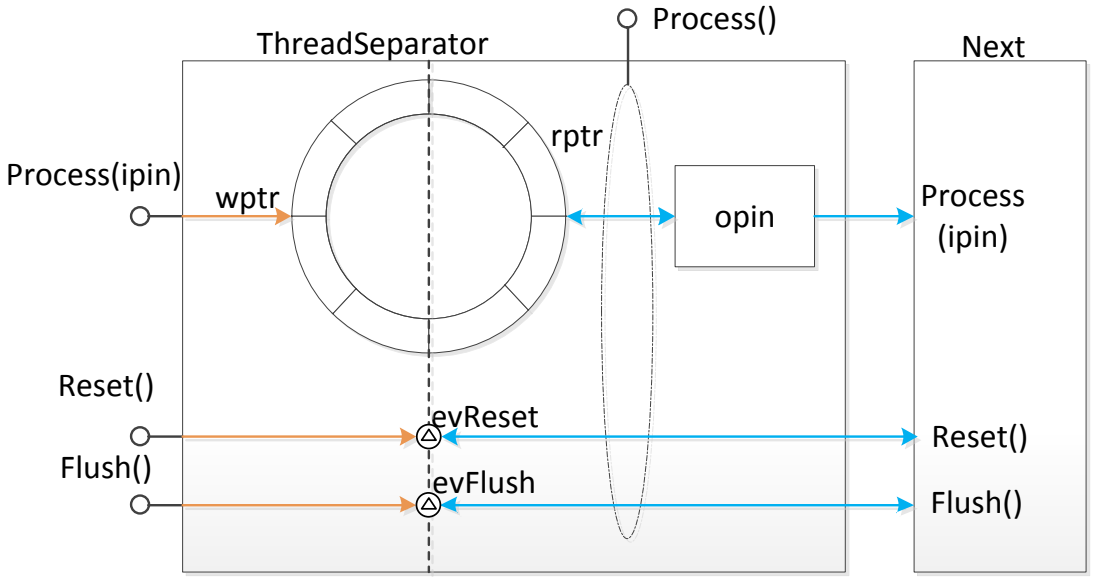

Figure 13. Internals of TThreadSeparator.

Chapter 3.
Stock Bricks

This section describes some brick modules that are shipped with Sora SDK Verion 1.6. The explanation in this section is meant to be brief. Please refer to the open source code (stockbrk.hpp) for the implementation details.

3.1 Basic sources and sinks

3.1.1 TMemSamples

TMemSamples loads I/Q samples from a memory buffer and pushs them downstream.

Brick type: Source

Required facade: CF_MemSamples, CF_Error

```
class CF_Error {
    FACADE_FIELD(ulong, error_code );       /* error code for the graph */
};

class CF_MemSamples {
    FACADE_FIELD (COMPLEX16*, mem_sample_buf);  // pointer to the memory holds I/Q samples
    FACADE_FIELD (uint,       mem_sample_buf_size);     // the size of the memory buffer in unit of bytes
    FACADE_FIELD (uint,       mem_sample_start_pos);    // the start sample to process in unit of samples
    FACADE_FIELD (uint,       mem_sample_count);        // # of samples in buffer
    FACADE_FIELD (uint,       mem_sample_index);        // index of sample being processed
};
```

Out port: COMPLEX16, 4

3.1.2 TRxStream

TRxStream reads I/Q samples from a RxStream and pushs them downstram.

Brick type: Source

Required facade: CF_RxStream, CF_Error

```
class CF_RxStream {
    FACADE_FIELD (SORA_RADIO_RX_STREAM*, rxstream_pointer );    // pointer to the
                                                                // RxStream object
    FACADE_FIELD (FLAG,                  rxstream_touched );
};
```

Out port: COMPLEX16, 28

3.1.3 TDropAny

TDropAny simply discards any input data.

Brick type: Sink

Required facade: CF_VOID

In port: Any type, 1

3.2 Basic filters

3.2.1 TDCRemove

TDCRemove substracts a constant offset (DC) from the input I/Q sample stream.

Brick type: filter

Required facade: CF_VecDC

```
class CF_VecDC {
      FACADE_FIELD (vcs, direct_current );    // the DC value
};
```

In port: COMPLEX16, 28

Out port: COMPLEX16, 4

Remarks: TDCRemove simply substracts DC from the sample stream. It assumes another module will estimate DC value and store it in CF_VecDC::direct_current.

3.2.2 TDCEstimator

TDCEstimate estimates the DC component of the input stream and stores the result in CF_VecDC::direct_current.

Brick type: filter

Required facade: CF_VecDC

In port: COMPLEX16, 4

Out port: COMPLEX16, 4

Remarks: TDCEstimator computes the DC component and adds the DC estimation into the DC value stored in CF_VecDC::direct_current. The following equation illustrates the operator,

$$\mathrm{DC} = \mathrm{DC} + \delta/4,$$

where δ is the DC estimation.

Chapter 4.
IEEE 802.11b Tranceiver

This section describes the brick implementation of an IEEE 802.11b transceiver. Figure 14 and Figure 15 show the processing graph of an 802.11b transmitter and receiver respectively.

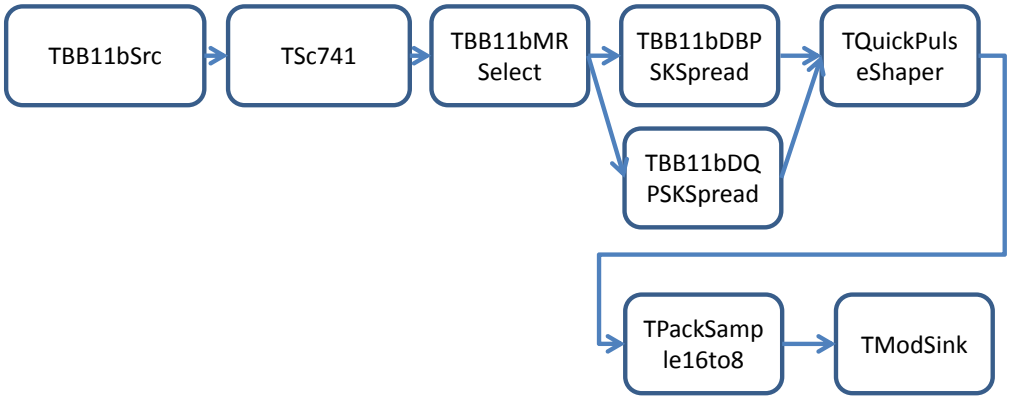

Figure 14. Sample 802.11b transmitter processing graph.

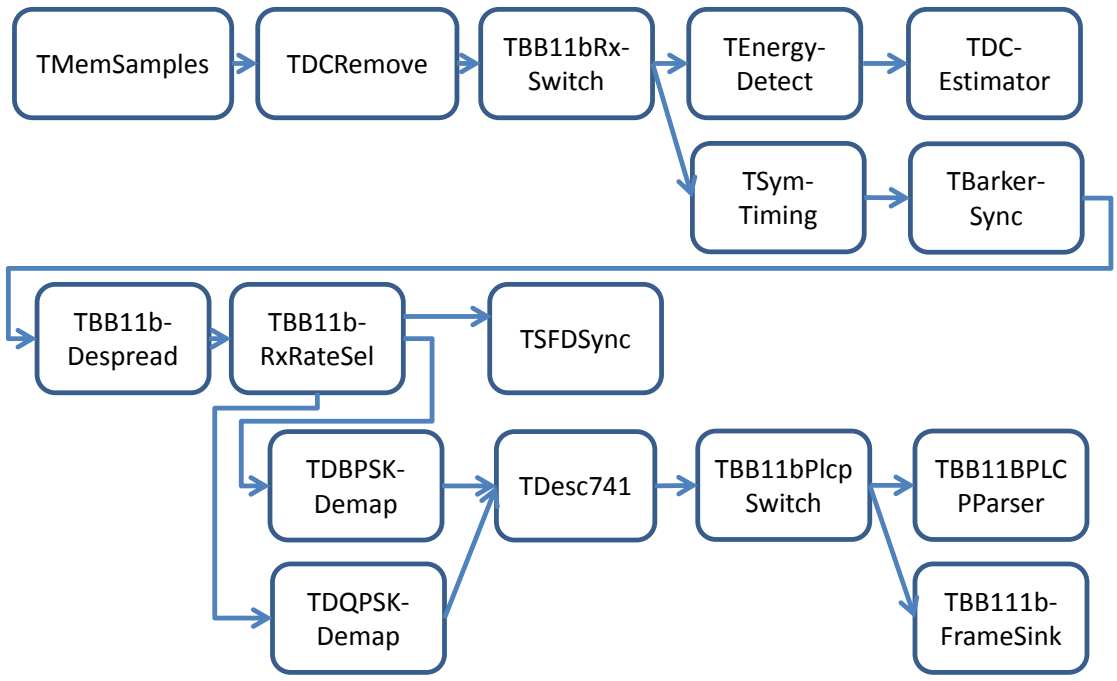

Figure 15. Sample 802.11b receive processing graph.

4.1 Transmitter modules

4.1.1 TBB11bSrc

TBB11bSrc takes a MAC frame from graph context, adds preambles and PLCP header, and pushes the PLCP service data unit (PSDU) downstream.

Brick type: source

Required facades: CF_11bTxVector, CF_TxFrameBuffer, CF_Error

```
class CF_11bTxVector {
      // INPUT
      FACADE_FIELD(ushort, frame_length );  /* 1-4095*/
      FACADE_FIELD(uchar, preamble_type);   /* 0=LONG, 1=SHORT*/
      FACADE_FIELD(uchar, mod_select);      /* 0=CCK, 1=PBCC */
      FACADE_FIELD(ulong, data_rate_kbps ); /* data rate in kbps */
      FACADE_FIELD(ulong, crc32 );              /* CRC32 for the frame */
};

class CF_TxFrameBuffer {
      FACADE_FIELD(uchar*, mpdu_buf0 );         // pointer to the first memory block of a MAC frame
      FACADE_FIELD(ushort, mpdu_buf_size0 );  // the size of the first memory block
      FACADE_FIELD(uchar*, mpdu_buf1 );         // pointer to the second memory block of a MAC frame
      FACADE_FIELD(ushort, mpdu_buf_size1 );  // the size of the second memory block
};
```

Out port: uchar, 1

4.1.2 TSc741

TSc741 scrambles the input byte stream.

Brick type: filter

Required facade: CF_ScramblerSeed

```
class CF_ScramblerSeed
{
      FACADE_FIELD (UCHAR, sc_seed);
};
```

In port: uchar, 1

Out port: uchar, 1

4.1.3 TBB11bMRSelect

TBB11bMRSelect transfers the input bytes to its two outports based on the modulation rate specified in CF_11bTxVector:: data_rate_kbps.

Brick type: filter

Required facade: CF_11bTxVector

```
class CF_11bTxVector {
      // INPUT
      FACADE_FIELD(ushort, frame_length );  /* 1-4095*/
      FACADE_FIELD(uchar, preamble_type);   /* 0=LONG, 1=SHORT*/
      FACADE_FIELD(uchar, mod_select);      /* 0=CCK, 1=PBCC */
      FACADE_FIELD(ulong, data_rate_kbps ); /* data rate in kbps */
      FACADE_FIELD(ulong, crc32 );              /* CRC32 for the frame */
};
```

In port: uchar, 1

Out port 0: uchar, 1

Out port 1: uchar, 1

4.1.4 TBB11bDBPSKSpread

TBB11bDBPSKSpread maps the input bytes to 8 symbols using DBPSK, and spreads the symbols using Barker code.

Brick type: filter

Required facade: CF_DifferentialMap

```
class CF_DifferentialMap {
      FACADE_FIELD(uint,  last_phase );
      // DBPSK: 0 - 00; pi - 01;
      // DQPSK: 0 - 00; pi - 01; pi/2 - 10; pi*3/2 - 11
};
```

In port: uchar, 1

Out port: COMPLEX8, 88

4.1.5 TBB11bDQPSKSpread

TBB11bDQPSKSpread maps the input bytes to 4 symbols using DQPSK, and spreads the symbols using Barker code.

Brick type: filter

Required facade: CF_DifferentialMap

In port: uchar, 1

Out port: COMPLEX8, 44

4.1.6 TQuickPulseShaper

TQuickPulseShaper upsamples the input symbol stream by four times and shapes it with a raised-cosine filter.

Brick type: filter

Required facade: CF_VOID

In port: COMPLEX8, 1

Out port: COMPLEX16, 4

4.1.7 TPackSample16to8

TPackSample16to8 converts a sample stream represented by COMPLEX16 to COMPLEX8.

Brick type: filter

Required facade: CF_VOID

In port: COMPLEX16, 8

Out port: COMPLEX8, 8

4.1.8 TModSink

TModSink collects modulated signal samples in a memory buffer.

Brick type: sink

Required facade: CF_TxSampleBuffer, CF_Error

```
class CF_TxSampleBuffer {
      // INPUT
      FACADE_FIELD(COMPLEX8*, tx_sample_buf );        // pointer to the memory holding the samples
      FACADE_FIELD(uint,      tx_sample_buf_size );   // the size of the memory buffer
      // OUTPUT
      FACADE_FIELD(uint,      tx_sample_cnt );        // number of samples in the buffer
};
```

In port: COMPLEX8, 8

4.2 Receiver modules

4.2.1 TSymTiming

TSymTiming recovers the best symbol timing. TSymTiming assumes the input sample stream is four times oversampled and it will perform four times decimation in the output stream. TSymTiming implements a simple early-later sampling detector to adaptively adjust the symbol timing estimation (http://cnx.org/content/m10485/latest/).

Brick type: filter

Required facade: CF_VOID

In port: COMPLEX16, 28

Out port: COMPLEX16, 1

4.2.2 TEnergyDetect

TEnergyDetect computes the average power of the input sample stream and compares it to a predefined threshold. If the average power is higher than the threshold, it will set CF_11CCA::cca_state to CF_11CCA::power_detected. TEnergyDetect will set

CF_Error::error_code to E_ERROR_CS_TIMEOUT if the average power remains lower than the threshold for a certain amount of time.

Brick type: filter

Required facade: CF_11CCA, CF_Error

```
class CF_11CCA {
public:
      typedef enum {
            power_clear = 0,
            power_detected
      } CCAState;


      FACADE_FIELD (CCAState, cca_state     );     // power detection state
      FACADE_FIELD (uint,     cca_pwr_threshold ); // the power threshold
      FACADE_FIELD (uint,     cca_pwr_reading   );  // the last power reading
      FINL void OnPowerDetected() { }                    // raised if the sample stream is detected
                                                         // in high power state
};
```

In port: COMPLEX16, 4

Out port: COMPLEX16, 4

4.2.3 TBarkerSync

TBarkerSync synchronizes to the boundary of a Barker sequence.

Brick type: filter

Required facades: CF_Error

In port: COMPLEX16, 1

Out port: COMPLEX16, 1

4.2.4 TBB11bDespread

TBB11bDespread dispreads Barker sequence.

Brick type: filter

Required facades: CF_VOID

In port: COMPLEX16, 11

Out port: COMPLEX16, 1

4.2.5 TSFDSync

TSFDSync searchs the incoming symbol stream to find SFD. When SDF is found, the receiver has synchronized to its byte boundary.

Brick type: sink

Required facades: CF_DifferentialDemap, CF_11bRxMRSel, CF_Descramber, CF_Error

```
class CF_DifferentialDemap {
	FACADE_FIELD(COMPLEX16,  last_symbol );
	// DBPSK: 0 - 00; pi - 01;
	// DQPSK: 0 - 00; pi - 01; pi/2 - 10; pi*3/2 - 11
};

class CF_11bRxMRSel {
public:
	typedef enum {
		rate_sync = 0,
		rate_1m,
		rate_2m
	} RxRateState;

	FACADE_FIELD (RxRateState, rxrate_state	);	// multi-rate selector
};

class CF_Descramber
{
	FACADE_FIELD(UCHAR, byte_reg); // for 802.11b
};
```

In port: COMPLEX16, 1

4.2.6 TDBPSKDemap

TDBPSKDemap demaps DBPSK modulated symbols into bits.

Brick type: filter

Required facades: CF_DifferentialDemap

In port: COMPLEX16, 8

Out port: uchar, 1

4.2.7 TDQPSKDemap

TDQPSKDemap demaps DQPSK modulated symbols into bits.

Brick type: filter

Required facades: CF_DifferentialDemap

In port: COMPLEX16, 4

Out port: uchar, 1

4.2.8 TDesc741

TDesc741 performs descrambling on the input byte stream.

Brick type: filter

Required facades: CF_DescamberSeed

In port: uchar, 1

Out port: uchar, 1

4.2.9 TBB11bPlcpParser

TBB11bPlcpParser parses the PLCP header defined by 802.11b standards.

Brick type: sink

Required facades: CF_11bRxVector, CF_Error, CF_11bRxMRSel, CF_11RxPLCPSwitch

```
class CF_11bRxVector {
	FACADE_FIELD(ushort, frame_length );   /* 1-4095*/
	FACADE_FIELD(ulong,  data_rate_kbps ); /* data rate in kbps */
	FACADE_FIELD(ulong,  crc32 );              /* CRC32 for the frame */
};

class CF_11RxPLCPSwitch {
public:
	typedef enum {
		plcp_header = 0,
		plcp_data
	} PLCPState;

	FACADE_FIELD (PLCPState, plcp_state );
};
```

In port: uchar, 6

4.2.10 TBB11bFrameSink

TBB11bFrameSink collects a MAC frame from the processing graph and stores the bytes into a memory buffer. TBB11bFrameSink also computes the CRC32 of the demodulated bytes and compares it to the CRC32 value embedded in the frame check sequence (FCS).

Brick type: sink

Required facades: CF_RxFrameBuffer, CF_Error, CF_11bRxVector

```
class CF_RxFrameBuffer {
    // INPUT
    FACADE_FIELD(uchar*,  rx_frame_buf );          // pointer to the memory to hold the bytes
    FACADE_FIELD(uint,    rx_frame_buf_size );     // size of the memory buffer
    // OUTPUT
    FACADE_FIELD(uint,  rx_frame_len );            // number of bytes have been stored in the buffer
};
```

In port: uchar, 1

4.2.11 TBB11bPlcpSwitch

TBB11bPlcpSwitch transfers the input byte to its two outports based on CF_11RxPLCPSwitch::plcp_state. An 802.11b PLCP parser should be connected to TBB11bPlcpSwitch outport 0 to parse the plcp header; while its output 1 should connect to a brick receiveing the data bytes of a frame.

Brick type: Demux2

Required facade: CF_11RxPLCPSwitch

In port: uchar, 1

Out port0: uchar, 1

Out port1: uchar, 1

4.2.12 TBB11bRxRateSel

TBB11bRxRateSel transfers the input symbol to its three outports based on CF_11bRxMRSel::rxrate_state. The outport 0 connects to a SFD synchronizer; outport 1 connects DBPSK demapper; and outport 2 connects DQPSK demapper.

Brick type: Demux3

Required facade: CF_11bRxMRSel

In port: COMPLEX16, 1

Out port0: COMPLEX16, 1

Out port1: COMPLEX16, 1

Out port2: COMPLEX16, 1

4.2.13 TBB11bRxSwitch

TBB11bRxSwitch transfers the input samples to its two outports based on CF_11CCA::cca_state. Its outport 0 connects to a subgraph performing CCA and its outport 1 should connect to a subgraph that demodulates the signal.

Brick type: Demux2

Required facade: CF_11CCA

In port: COMPLEX16, 4

Out port0: COMPLEX16, 4

Out port1: COMPLEX16, 4