Chapter 4

GNU Radio Architectural Changes

`arch/gnuradio-chapter.tex,v 1.62 2006/06/10 02:17:37 dlapsley Exp`

4.1 Introduction

GNU Radio is a powerful and flexible framework that allows the rapid composition of Software Defined Radios (SDRs) using complex signal processing *flow_graphs* that are composed of simple signal processing building blocks. We propose some extensions to the GNU Radio baseline architecture that will improve support for packet radio protocols. The changes proposed can be implemented in an incremental fashion as separate modules on the existing GNU Radio framework and will have no impact on existing software or functionality.

The main extension proposed is the introduction of a new block type called the *message-block* or *m-block*. This block provides a number of new capabilities not currently available in GNU Radio:

- Processing of data *messages* (arbitrarily sized blocks of data for transport coupled with metadata).
- Processing of timing information and maintenance of timing synchronization across a *flow_graph* and between threads or processes.
- Handling and generation of signals from and to external entities.
- Processing of commands from external entities and reporting of status to those same entities.
- Integration with a new top level scheduler to support hierarchical, quasi-real-time scheduling.
- Block aggregation and support for hierarchical scheduling.

The *m-block* also integrates readily with existing GNU Radio blocks and scheduling algorithms. In addition to the new block type, we propose the addition of the following elements/capabilities:

- Message-based Scheduling.
- Standardized time transfer mechanism.
- Dynamic re-configurability.
- Support for control interfaces.
- Hierarchical, quasi-real-time, hybrid scheduler.
- Memory Management to support the functionality above.

©2006 BBN Technologies Corp.

This chapter describes the proposed extensions to GNU Radio. We begin by describing architectural requirements to support packet radio protocols. We then discuss the baseline GNU Radio architecture and detail proposed extensions to the baseline GNU Radio architecture. We then discuss a high-level roadmap that shows how proposed extensions could be incrementally added as separate modules on the GNU Radio architecture. We conclude the chapter with a summary of the changes required in the GNU Radio architecture to support packet radio protocols.

We believe these extensions to GNU Radio's architecture can be added in an incremental manner and will be useful to the community in providing support for packet radio protocols. We encourage and greatly appreciate any comments or feedback from the community on these proposed extensions.

4.2 Motivation

The GNU Radio framework provides an excellent environment to create and run complex signal processing functions, and to connect them to the RF world. GNU Radio has already demonstrated its expressiveness and versatility by rapidly implementing a number of very complicated signal processing programs, for example a High Definition television receiver. GNU Radio already provides a strong foundation for radio development that enables academics, industry, and hobbyists to collaborate and innovate effectively.

The GNU Radio project is currently working on — among other goals — implementing sophisticated modulators and demodulators for digital wireless telephony and wireless packet communications. We believe that the GNU Radio approach could be augmented to provide the flexibility and control required by higher layer networking protocols. This extension will further broaden the user community, and will accelerate collaboration in the emerging Software Defined Radio area.

The proposed extensions to GNU Radio are not many, but they are important for packet radio. The Media Access Control (MAC) layer needs low-latency transmission control – faster than the FIFO processing currently implemented in GNU Radio *flow_graphs*. The extensions allow a flow to execute to fill a buffer, so that the sample data is pre-computed and ready to go upon receipt of a signal. The extensions implement a signaling mechanism that quickly delivers signals to processing blocks, either from other blocks or from programs running outside the GNU Radio context. Some MACs require tight timing and time-tagging, and that capability is provided by the proposed extensions. Finally, higher layers like to track and operate on collections of bytes together, and in hierarchies of collections. The network layer thinks of "packets", and the link layer considers "frames", either of which may be composed of multiples of the other. The extensions we propose incorporate the ability to manipulate and tag buffers to meet those needs.

Transmission priority is also of concern to the network and link layers. Quality of Service implementations allow the network to match interfaces with different bitrates and loading to each other, and to allow higher-priority packets to get through, even though lower-priority packets arrived at the interface earlier. This priority needs to reach down all the way to the transmitter to satisfy the latency needs of the network layer. If not accounted for by the physical layer, a large lower-priority packet already in transmission, for example, might occupy the channel, preventing a higher priority packet from being transmitted in a timely fashion.

4.3 Requirements

4.3.1 MAC Layer Requirements

On behalf of the MAC, GNU Radio must meet the following requirements:

REQ 4.1 Expose (describe) the capabilities and characteristics of GNU Radio (software and hardware) according to predictable conventions.

REQ 4.2 Provide event signaling (notification) as directed by a client or user system of the GNU Radio.

 ©2006 BBN Technologies Corp.

REQ 4.3 Accept control directives of the client or user systems of the GNU Radio, including dynamic changes in operating parameters.

REQ 4.4 Interrupt and resume device operations per MAC direction.

REQ 4.5 Accept frames from the MAC, convert to symbol streams, and transmit, with appropriate signaling, including symbol and frame synchronization.

REQ 4.6 Capture raw radio frequency samples from a physical radio device, process samples appropriately and convert to data symbols, convert to bytes, and then forward to the MAC, along with appropriate physical layer statistics.

4.3.2 Derived Requirements

Some MAC protocols do not make use of a sense of time. Indeed, it is conceivable that a CSMA/CA MAC may be designed to avoid requiring any independent timing capabilities of the Radio Device. However, as a practical matter, a time reference and timing capabilities are indispensable for flexibility of MAC development and good performance. Therefore, we list the following as derived requirements, while recognizing that there is room for discretion in where and how the requirements are satisfied:

REQ 4.7 GNU Radio must be able to share a time reference with the MAC, and be able to synchronize with that reference with [TBD] accuracy.

REQ 4.8 GNU Radio must have a timer class and an ability to perform time-based activity scheduling, with sufficient granularity to meet MAC protocol needs.

REQ 4.9 Scheduling to meet timing objectives supplied by MAC protocols (or by other sources) will require the scheduler to take accurate account of latencies in the processing chain throughout the radio device. GNU Radio will have to supply mechanisms to collect and utilize latency information.

4.3.3 Qualitative Requirements

Finally, we recognize the following qualitative goals as effective requirements:

REQ 4.10 Minimize unnecessary changes to GNU Radio and GNU Radio framework.

REQ 4.11 Allow flexibility for future GNU Radio evolution and GNU Radio hardware alternatives (*e.g.,* Ethernet vs. USB for USRP connection).

REQ 4.12 Blocks must be capable of accepting and responding to control commands from one or more external entities. They must do so through a defined control interface.

REQ 4.13 Blocks must be capable of reporting operational and status information to one or more external entities. They must do so through a defined control interface.

REQ 4.14 Blocks that are required to report status information to external entities must implement functionality to report this information via a standard control interface.

REQ 4.15 For blocks whose operation depends on accumulated state (*eg,* filters with history, blocks with embedded finite state machines), control actions that interrupt processing or otherwise alter the environment dynamically may invalidate dependencies on state. Such stateful blocks must have the ability to restore consistent internal state after configuration changes.

©2006 BBN Technologies Corp.

4.4 Definitions

Table 4.1 lists the definitions of some common terms that are used throughout this chapter.

Term	Definition
data	the information content to be transferred between two hosts. Data is transformed as it passes through a GNU Radio *flow_graph* or through a set of *m-block*s. Examples of data include packets, bytes, bits, and raw radio frequency I- and Q-samples.
metadata	structured encoded information that describes the characterics of its associated data and the ways in which that data should be transformed by any GNU Radio blocks or *m-block*s. Metadata is typically not transferred between hosts. Examples of metadata include transmission power, modulation scheme, and FEC type.
message	the smallest unit of information that can be processed by an *m-block*. A message consists of data and metadata (control data that describes how data should be transformed during the transmit/receive processing — e.g. type of modulation to use, power level at which to transmit, received power, received modulation type, etc.).
block	the baseline GNU Radio block (*gr_block*).
m-block	a new version of the GNU Radio block that is capable of processing messages.
flow_graph	a *flow_graph* made up of only baseline GNU Radio blocks.

Table 4.1: Definitions of common terms

4.5 GNU Radio Baseline

This subsection describes the GNU Radio architecture before the proposed extensions.

GNU Radio is an extensible free software framework for the creation of software radios. The GNU Radio framework also incorporates software that supports the easy integration of a number of hardware modules so that radio signals may be received from, transmitted to or exchanged with other GNU Radio-based software radios or conventional radio systems.

GNU Radio uses a modular, block-based architecture that utilizes a hybrid Python/C++ programming model. The combination of Python and C++ provides a convenient and high performance platform for developers to use in the development of software radio systems. Functionality that requires CPU-intensive processing is implemented in C++ for high performance, while functionality that involves complex interactions between blocks is implemented in Python.

One of the features of the GNU Radio framework is the extensive library of pre-defined and tested functional blocks. These blocks provide signal processing functionality, encapsulate sources and sinks of data and provide simple type conversions. The blocks are written in C++ and have a (typically) automatically generated Python "wrapper" or interface that allows them to be manipulated, connected and utilized in Python. New blocks can easily be added to the block library, indeed the GNU Radio community strongly encourages the addition of new blocks implementing new functionality or improved performance.

 ©2006 BBN Technologies Corp.

GNU Radio processing blocks may be hooked together and run from a python program. The python program provides a framework for the processing blocks to communicate via buffers, and also provides a simple scheduler whereby the various processing blocks making up a radio transmitter or receiver are executed sequentially depending on the availability of inputs to the block meeting its processing criteria to execute a single processing iteration. The scheduler in the current GNU Radio system relies on a steady stream of data input to the collection of blocks to cause the blocks to run and produce output.

A GNU Radio software radio typically consists of the following elements:

Sources: A GNU Radio software radio will have at least one source. Each source is the head of a processing chain or *flow_graph*. An example of a GNU Radio source is the Universal Software Radio Peripheral (USRP) radio. This is a radio front end that connects to a computer via a USB 2.0 bus. Software drivers are available that allow the integration of a USRP into a GNU Radio program.

Sinks: A GNU Radio software radio will have at least one sink. Each sink is the tail of a *flow_graph*. An example of a sink is a sound card.

Flow graphs: A GNU Radio software radio will have a *flow_graph* that links together each source and sink pair as well as any intermediate blocks that are required to transform the data stream from a source into a format that is understandable by a sink. For example, converting an FM radio signal that is received by a USRP into an audio signal that can be played through a sound card.

Schedulers: A scheduler is associated with each active *flow_graph*. Each scheduler is responsible for moving data through its *flow_graph*. A scheduler iterates through the blocks in a *flow_graph*, identifies blocks that have sufficient data on their input (s) and sufficient space on their output (s) to be able to process data and then triggers the processing function for those blocks. The scheduler in the current GNU Radio system relies on a steady stream of data input to the collection of blocks to cause the blocks to run and produce output.

4.5.1 Scheduling and Memory Management

GNU Radio processes data as a stream of homogeneous items. These items are typically floats, doubles or complex values but may include any C++ element for which `memcpy` is a valid constructor. This includes many structures and classes. The GNU Radio scheduler breaks each data stream into chunks and feeds these chunks one at a time into each block in the *flow_graph* using the mechanism described in Algorithm 4.5.1 where each block provides a `general_work` function that is called by the scheduler and is passed one chunk of data per input. The size of the chunks depends on many factors, in particular, the type of block. The size of the chunks can also vary from call to call for a given block. A block can have zero or more input streams and zero or more output streams, but not both zero input and output streams. The rate of data flowing into the block can vary between different input streams, but the rate of data flowing out of a block must be constant across all output streams.

Figure 4.1 provides a notional illustration of the working of the GNU Radio scheduler, omitting detail. In the bottom of the figure is a *flow_graph*, consisting of three blocks, labeled *1, 2, 3*, and connections between output and input ports, three of which are labeled as *a, b, f*. In the top of the figure is a notional illustration of the memory structures utilized to contain the data that is passed across connections, in other words, data that is emitted from output ports and read into input ports. Every block declares fixed unit sizes on each of its input ports that are necessary to support one atomic operation of the block's processing task (the `general_work()` method). The block also has a fixed unit size for the data conveyed to its output ports in one atomic operation. These unit sizes are known to the scheduler at initialization, and the scheduler arranges memory mappings to hold the data that will be exchanged across connections. For the duration of the *flow_graph* configuration, the mappings are static: both the location and size of memory allocations are fixed. The scheduler is also responsible for memory management during runtime, that is, arranging for the storage and access of intermediate data products as transmission data flows through the signal processing chain. The scheduler exercises this management function in part via delegation to helper classes.

©2006 BBN Technologies Corp.

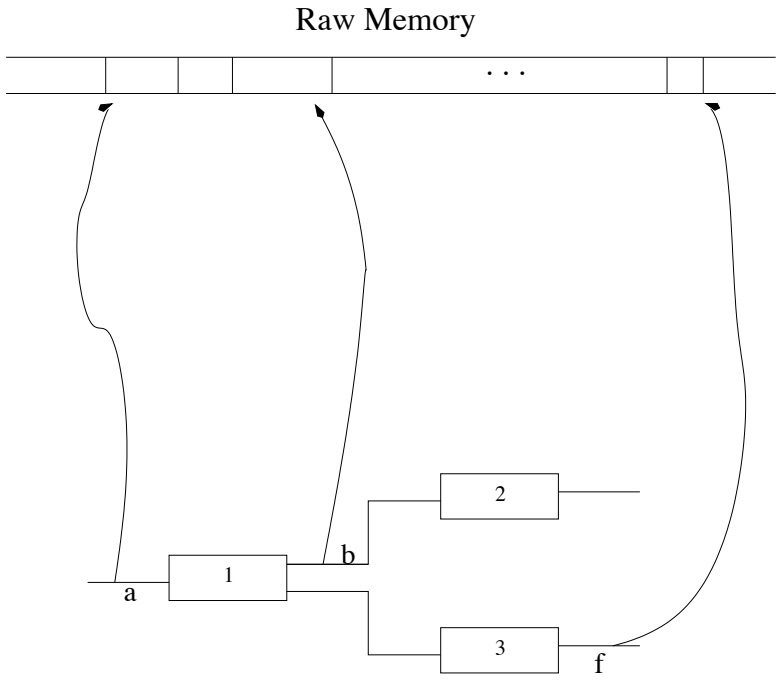

Figure 4.1: Simplified View of the GNU Radio Scheduler

The scheduler is essentially a cyclic poller, calling each block in turn to perform its processing function, always cycling in the same order. The basics of its operation can be conveyed by the pseudo-code in Algorithm 4.5.1.

Algorithm 1 GNU Radio Scheduling loop.

```
while enabled and nalive > 0 do
  for i = 1, 2, ..., #blocks do
    if sufficient room in output port buffers for block i and data at input ports of block i then
      invoke general_work() on block i ;
    end if
  end for
end while
```

This "systolic" scheduling is convenient for processing continuous streams of data such as broadcast radio and television transmissions. However, it is not ideal for the support of packet radio for the following reasons:

- Packet radio has real-time scheduling requirements across multiple threads of execution.
- Packet radio requires complex and dynamically varying metadata to describe the data transformations required at each processing block.
- Packet radio can significantly benefit from dynamically reconfigurable processing blocks. Dynamic reconfiguration is intertwined with scheduling and requires support from both the *m-block* scheduler and related blocks.

The primary memory management scheme in GNU Radio was designed to work with blocks that have very concrete specifications concerning the data units taken as input and produced as output in a single

 ©2006 BBN Technologies Corp.

processing invocation. In particular, there is an assumption of known and fixed memory requirements for each input and output datum. This design is well suited for signal processing blocks.

4.5.2 Packet Radio

Great strides have been made to support packet radio within the baseline GNU Radio framework. An example of this is the `gmsk2_pkt` module which supports the transmission of packets of data using Gaussian Minimum Shift Keying (GMSK). This module illustrates how powerful the GNU Radio framework is for composing complex signal processing chains. The `gmsk2_pkt` module has two classes: `gmsk2_mod_pkts` and `gmsk2_demod_pkts`. The `gmsk2_mod_pkts` class is used for transmission while the `gmsk2_demod_pkts` class is used for reception.

The `gmsk2_mod_pkts` class allows a user to feed a packet of data into a signal processing chain via the use of a `send_pkt` primitive. This primitive inserts the user packet into a thread safe message queue which is connected to a signal processing graph that uses a hiearchical block construct to implement GMSK modulation. The message queue converts the data packet into a stream of data samples which are then modulated by the GMSK modulation *flow_graph* and exit the *flow_graph* as a series of complex baseband symbols.

The `gmsk2_demod_pkts` class allows a user to register a callback function that will be called whenever a demodulated packet is received. The callback is called with the result of demodulating the samples. A *flow_graph* is constructed that uses a hierarchical block construct to implement GMSK demodulation. Raw complex baseband samples are fed into one end of the *flow_graph*, are demodulated and then stored in a thread safe message queue. A separate "watcher" thread continuously takes received packets from the head of the message queue, parses them and then forwards them to the pre-registered callback function. When there are no packets in the message queue, the watcher thread blocks and waits for the next packet to be received.

While these modules implement impressive functionality, there are still a few limitations with the `gmsk2_pkt` module:

- The module still uses stream-based scheduling and so does not have access to metadata that could be used to dynamically change signal processing block parameters on the fly.
- The module does not have a well-defined control interface through which individual blocks can be controlled, signaled, and report information to external entities.

4.5.3 Summary

The GNU Radio baseline architecture provides a powerful framework for building complex signal processing chains for Software Defined Radios. Great progress has been made in building support for packet radio within GNU Radio. However, there are still some limitations that we believe could be overcome with the addition of the proposed extensions to the GNU Radio baseline architecture.

4.6 Proposed Extensions to GNU Radio

This section presents the designs and plans for extending the GNU Radio architecture so that it is useable for packet radio protocols.

4.6.1 Overview

The proposed extensions are to be implemented as modules which are independent of the current modules. This will allow users to choose between using the new *m-block*s, the current GNU Radio blocks or a combination of the two. If a combination of *m-block*s and GNU Radio blocks are used, GNU Radio blocks must be encapsulated within *m-block*s(as described in section 4.8.2).

©2006 BBN Technologies Corp.

4.6.2 Combined data and metadata

The proposed extensions extend the GNU Radio stream-based paradigm to allow metadata to be associated with data and transported as discrete *messages* of information. The proposed architecture will define a standard format for the metadata and provides functionality to generate, transport, manipulate and parse this information.

4.6.3 Message Block

The proposed extensions include a new type of GNU Radio block. We call this block a *message*-block (or *m-block*). Information flows into or out of *m-block*s as messages that flow into or out of bi-directional *m-block* typed ports. These messages may communicate data, metadata, control information, status information, signals or a combination of these. *m-block*s process any control or signaling information that is sent to them and transform any data using information supplied within the associated metadata. Each port has an associated protocol class that specifies which messages may pass into or out of that port. This port typing ensures that only compatible ports are connected together. For a complete definition of port compatibility, see Section 4.9.5. Figure 4.2 illustrates the main features of *m-block*s.

In this section, we provide a high level view of the main features of *m-block*s. For a more formal specification of *m-block*s refer to Section 4.9.

*m-block*s also provide support for aggregation. *m-block*s may be enclosed within a single *m-block* and treated as a single entity. The enclosing *m-block* may provide interfaces that allow the transfer of messages from outside the *m-block* to the enclosed *m-block*s, and vice-versa. Multiple levels of aggregation may be used. The architecture places no limit on the number of levels of aggregation. We refer to *m-block*s contained within another *m-block*s as *component m-block*s and to *m-block*s that contain other *m-block*s as *aggregate m-block*s. We refer to an *m-block* that does not contain any other *m-block*s as a *simple m-block*.

The aggregation capability is extended to allow *m-block*s to contain collections of GNU Radio blocks or GNU Radio *flow_graph*s. In this case, in addition to providing interfaces to allow the transfer of information between the enclosed GNU Radio *flow_graph* and the outside world, the enclosing *m-block* also provides interworking functionality to convert between messages and streams of data samples that are input to or output from the enclosed GNU Radio *flow_graph*.

In GNU Radio, blocks are connected together through the use of a separate *flow_graph* structure which maintains topological information and provides a scheduler that runs in its own thread of execution. The proposed extensions have a slightly different model. The aggregation property of *m-block*s requires *m-block*s to maintain topological information. This eliminates the need for a corresponding *flow_graph* for *m-block*s. Instead, whenever a system comprising more than one *m-block* is used, there will always be a single top-level *m-block* that will provide the equivalent topological functionality of a GNU Radio *flow_graph*.

The *m-block* functionality that has been described so far, will all be implemented within a developer framework that will include a base *m-block* class. This class will include data structures for buffering incoming messages, and functionality to manage those buffers. A `send_message` primitive will be provided to allow a *m-block* to send messages through its ports. The base *m-block* class will also include methods to enable the addition/removal of component *m-block*s in a dynamic and thread-safe manner. Primitives will be provided to allow the connection of internal components of the *m-block* to each other and to external interfaces.

In order for a developer to create a new type of *m-block*, they simply inherit from the base class.

4.6.4 Message-based Scheduling

The proposed extensions use message-based scheduling. In this case, information is transported between *m-block*s as a series of discrete messages. Each message consists of metadata and optionally data. The metadata may be used to convey control information such as commands or signals. The metadata is also used to describe any data that is present in the message and how that data should be transformed. Examples of metadata include information such as timestamps, modulation schemes or power levels. The data portion

 ©2006 BBN Technologies Corp.

of the message carries the information to be transformed by the blocks in the *flow_graph*. Metadata may also be used to convey control information such as commands or notification messages.

A highly extensible format for the messages is provided by the developer framework.

4.6.5 Hierarchical, quasi-real-time, hybrid scheduler

The proposed extensions support the needs of time-knowledgeable, priority-based scheduling required for processing *m-block*s, as well as reconcile the interoperation between current GNU Radio *flow_graphs* and the new *m-block*s. This will be achieved through the use of *m-block*s and a hierarchical, quasi-real-time, hybrid scheduling scheme.

The scheduling algorithm is priority-based. Each *m-block* is assigned a priority which is equal to the priority of the highest priority message in its input buffer. The scheduler determines which *m-block* has the highest priority and then dequeus the highest priority message and invokes the `handle_message` callback.

The scheduling mechanism is also able to interoperate with GNU Radio *flow_graphs*. Any GNU Radio *flow_graphs* that are encapsulated within an *m-block* are evaluated within the `handle_message` callback. The enclosing *m-block* provides the necessary conversion functionality to convert messages into a format that the GNU Radio *flow_graph* will understand and vice-versa.

The developer framework provides all of the functionality described above in the form of methods that are encapsulated within base classes. Developers can make use of this functionality by inheriting from these base classes and calling on these methods.

4.6.6 Common Time Reference

A given system may contain multiple transmitters and/or receivers with a corresponding number of clocks. We assume that one receiver will act as a Master clock and that all of the other receivers will slave to that clock. We choose the sample clock of the Master receiver sample clock as the authoritative time reference that is used for all scheduling operations.

A primitive will be provided by the developer framework that will enable the developer to query the current value of the time reference.

4.6.7 Standardized time transfer mechanism

The proposed extensions provide a standard mechanism for transferring timing information between various parts of a *flow_graph*: every message that is processed by a collection of *m-block*s contains a list of entry/exit timestamps for blocks in the processing path.

This information can be used to calculate latency through individual blocks, and between different layers within an SDR protocol stack. This information can be used by the scheduler to guide the allocation of processing resources in order to ensure that timing requirements are met (if possible).

4.6.8 Dynamic reconfigurability

The proposed extensions provide facilities that allow the dynamic reconfiguration of an *m-block*. Primitives will be provided to allow the addition or removal of component *m-block*s to an *m-block* and to allow the connection of an *m-block*'s component *m-block*s to each other and/or to interface ports on the *m-block*. These primitives will support these operations in such a way that an active *m-block*, i.e., an *m-block* that is currrently processing data, can be reconfigured without corrupting the data flow.

4.7 Proposed Extensions Roadmap

We believe the proposed extensions can be implemented in an incremental manner as separate modules in the GNU Radio codebase. We believe that by implementing the extensions in the following order it is possible to

incrementally add functionality to the GNU Radio framework in a way that provides useable functionality at each stage.

1. Message block
2. Memory management
3. Message-based scheduling
4. Hierarchical, quasi-real-time, hybrid scheduling
5. Standardized time transfer mechanism
6. Dynamic re-configurability
7. Control Interfaces

This functionality will be implemented in C++. We allow for the possibility of subclassing in Python using SWIG's "director" feature.

4.8 Implementation Considerations

Here we flag a few issues that may concern implementers, and we list capabilities that appear desirable for for inclusion in the GNU Radio software base, but which may not be essential. We expect to provide these capabilities, or not, contingent upon resources available and priorities that emerge in the course of the project.

4.8.1 Message Block

In this section, we describe in more detail *m-block* functionality and how *m-block*s would are implemented. Each *m-block* port has an associated *protocol class* that specifies the type of the port and the messages that may enter or leave the port. In order to be connected to each other, two *m-block* ports must be compatible (see Section 4.9.5). An *m-block* may have zero or more bi-directional message ports. There may be multiple ports of the same protocol and ports can be uni-directional if needed.

Figures 4.2 and 4.3 show two different views of an *m-block*. Figure 4.2 shows the general form of an *m-block* while Figure 4.3 shows how *m-block*s can be used to encapsulate *flow_graphs* made up of existing GNU Radio blocks that utilize the baseline GNU Radio scheduler.

Each *m-block* has a processing callback function: `handle_message`. The `handle_message` callback is invoked externally and runs within the context of the invoking thread or process. All of an *m-block*'s processing is performed within `handle_message`: message timing functionality, data processing and metadata processing. The `handle_message` callback never blocks waiting for an external event but relies on the scheduler for external event notification. An instance of an *m-block* never has its `handle_message` callback invoked recursively, thus there are no internal reentrancy concerns.

An *m-block* has a priority that is equal to the priority of the highest priority message that is currently queued in the *m-block*. This priority is used by the scheduler when determining which *m-block* to schedule next. In general, the highest priority *m-block* will be scheduled next by the scheduler, which will be discussed in more detail in Section 4.8.2. The use of priorities allows the rapid propagation of high priority signals through a collection of *m-block*s.

Messages may vary in size and in the amount of processing required at a given *m-block*. For example, a message that contains a buffer of time-domain symbols that are to be spread by a particular *m-block*, will take much longer to process than a message that simply requires the addition of a padding byte by another *m-block*. In order to support real-time scheduling, a mechanism is required that will allow *m-block*s

 ©2006 BBN Technologies Corp.

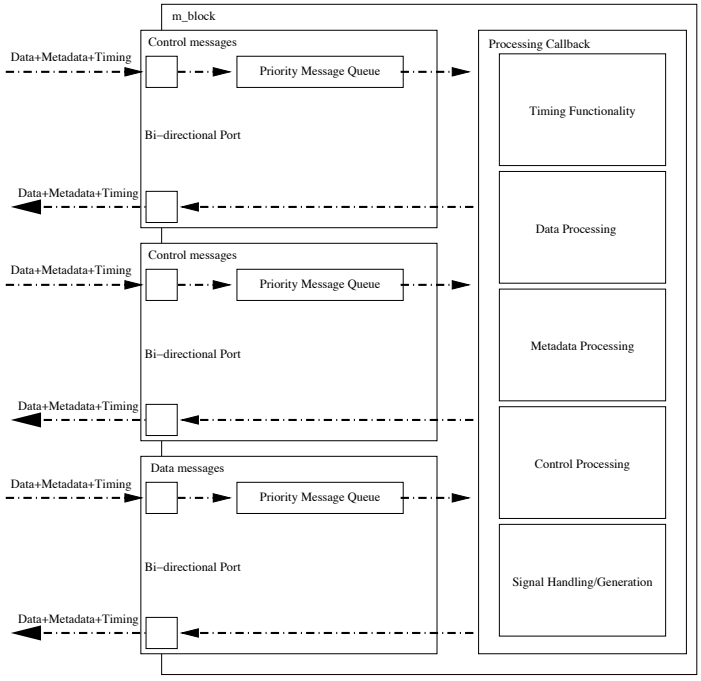

Figure 4.2: General Form of "m-block"

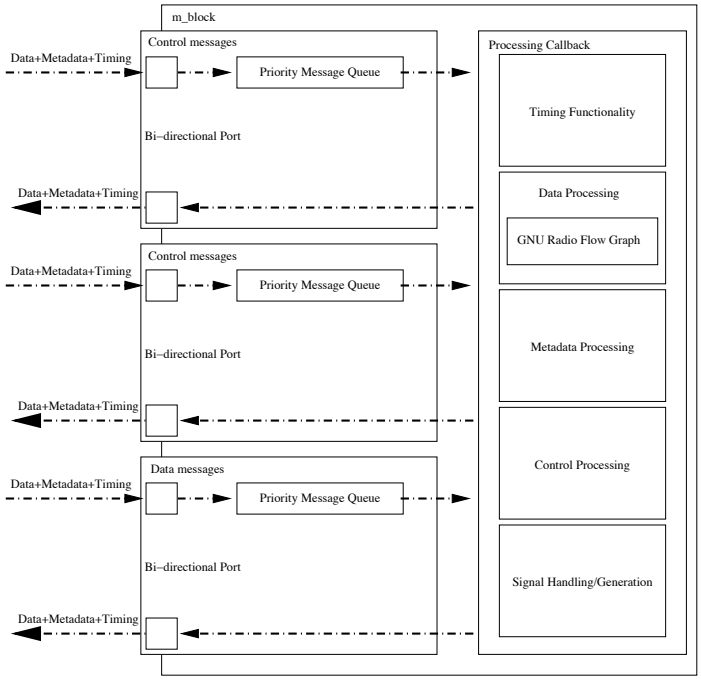

Figure 4.3: "m-block" with internal GNU Radio flow graph

©2006 BBN Technologies Corp.

to relinquish control of a processor after a certain number of processor cycles have been used. Most of the support for this is provided by the scheduler (see Section 4.8.2).

In the case of an *m-block* that contains a *flow_graph*, when a message is received that contains data to be processed by the *flow_graph*, the callback passes the relevant portion of the data to the *flow_graph* source. It then executes `gr_single_threaded_scheduler` (the C++ implementation of the existing *flow_graph* scheduler) using the current thread of execution. That is, we run the *flow_graph* scheduler from the callback.

The `gr_single_threaded_scheduler` will run until it walks all the incoming data through the graph to the final sink, at which point it returns. At this point, the callback would typically retrieve the data from the sink using some zero-copy technique, and send messages to one or more of its ports preserving the *m-block*'s encapsulation in the process.

4.8.2 Hierarchical, quasi-real-time, hybrid scheduler

4.8.2.1 Basic Mechanism

The *scheduler thread* continuously loops and attempts to schedule *m-block*s that are ready to run. The scheduler selects the highest priority block (the block with the highest priority message on its input queue) and invokes its callback to start the *m-block*'s processing. If no *m-block*s are ready for scheduling, the scheduler will go to sleep until it is woken up.

*m-block*s have the option to call a `yield` function from time to time to relinquish control of the processor and allow the scheduler to check for and process any *m-block*s that may have increased in priority while the *m-block* was processing. The scheduler then services any *m-block*s that have a higher priority than the yielding *m-block* before allowing it to continue processing.

4.8.2.2 Interoperability with GNU Radio Single Threaded Scheduler

Figure 4.4 shows how the new *m-block* scheduling scheme interworks with current GNU Radio scheduling. A chain of GNU Radio blocks is divided up into separate GNU Radio *flow_graphs* and then each *flow_graph* is encapsulated within an *m-block*. Within an *m-block*, data is moved through the GNU Radio *flow_graph* using a GNU Radio scheduler that runs within its own thread of execution. However, information transfer between *m-block*s is regulated by the top level scheduler which signals each *m-block* in turn when it is time for the *m-block* to process the message at the head of the priority queue for each of its inputs, and then pass the processed messages on to its downstream neighbor *m-block*s.

The time transfer mechanism described in section 4.6.7 is used to collect and exchange timing information between elements within a *flow_graph*. The scheduler has access to this timing information and so is able to allocate processing resources based on the measured latencies between endpoints.

Additionally, the scheduling mechanism relies on message priority. Priority markings within a "user range" may be applied externally to the messages at the origin. This marking mechanism simplifies inter-block signaling implementation through the injection of "signaling messages" carried at a higher priority reserved to the *m-block* scheduling mechanism itself. It will be seen later that a unified message handling loop which subsumes both regular data and signals simplifies the implementation of the proposed features.

This architecture is very flexible. The subdivision and encapsulation of GNU Radio *flow_graphs* can be varied manually or through some higher level automatic control, and the way in which processing resources are allocated to each *m-block* can be varied. This architecture also allows for the interworking of GNU Radio blocks and *flow_graphs* with *m-block*s.

The proposed *m-block* executes under the control of the new scheduler. The *m-block* consumes messages by defining a callback function. The scheduler processes the *m-block*'s input message queue and selects the highest priority message to present to the *m-block*'s callback.

 ©2006 BBN Technologies Corp.

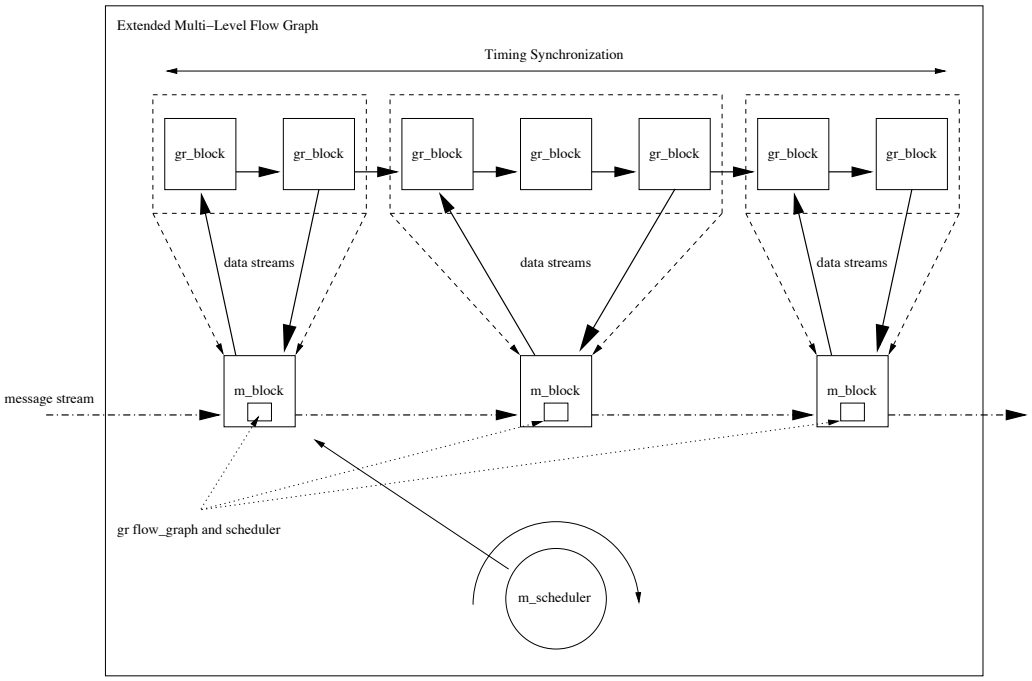

Figure 4.4: Extended higher level scheduler

4.8.3 Memory Management

In extending the GNU Radio framework to support packet radio protocols, we propose to supply additional infrastructure for handling the message data that the radio will exchange with these protocols. Such data:

- may be of variable length
- is accompanied by metadata
- requires persistence as the message is processed by the radio
- may contain elements that change as the message is processed in the radio.

This message data will be handled by *m-block*s which will have the ability to:

- Associate metadata with the portion of the message data that is destined for transport.
- Parse the contents of metadata and control data, and manipulate the contents.
- Decouple transport data and marshall it for submission to GNU Radio *flow_graphs*.

In addition, to support the extended scheduling mechanisms and the transfer of data among extended processing blocks, and to help minimize the opportunities for coding errors, the extensions we propose to GNU Radio include supporting utilities for buffer and buffer queue management. Functions performed by these utilities include:

- automatic sizing and resizing
- safe copying and concatenating
- easy header manipulation and handling of metadata

©2006 BBN Technologies Corp.

- zero-copy efficiency
- reference counting.

4.8.4 Data and Metadata

An open and flexible message structure will be used to communicate information between *m-block*s and with external entities. Table 4.2 shows some of the fields available within a single message. Each message contains two types of information:

Message/Metadata Information: This includes standard fields that are used to convey information about the message (message version, timestamps, handle etc.) as well as fields that specify how the message is to be handled by the scheduler (priority), how the message is to be handled by *m-block*s (segmentation structure and signal) as well as commands and/or notifications from external entities. Table 4.2 provides a more detailed description of each of the fields.

Data portion: This portion of the message carries the data. The data could be in the form of user bytes, forward error corrected user bytes, modulated data, spread data etc. As a message makes its way through a *flow_graph*, the data is transformed by the blocks along the way.

Messages provide a powerful way to dynamically program blocks on the fly. The also provide a fundamental mechanism for transporting data, metadata, signals, notifications and control information between *m-block*s.

Field	*Description*
Message Handle	Unique identifier for this message.
Priority	This indicates the priority of the message.
Signal	This is an event name that is used to control state transitions within the Finite State Machines of *m-block*s that process this message.
Timestamps	A table that carries the entry and exit timestamps for blocks in the processing path. The key of the map uniquely identifies the processing block at which the timestamps were recorded. Note that not all blocks in the processing path need to record their timestamps.
Data portion	This section contains data to be operated on by the blocks in a processing *flow_graph*.

Table 4.2: Sectional Message Fields

4.8.5 Protocol Classes

Protocol classes specify the types of messages that may go into or come out of a port. Protocol classes also have directionality: they specify the direction in which a message is permitted to travel ("in" or "out" of a port). Each *m-block* port has an associated protocol class and is deemed to be either *unconjugated* or *conjugated*. *Unconjugated* ports have the same directionality as their associated protocol class, while *conjugated* ports have the reverse directionality. In order to bind two ports together for message passing, the ports must be compatible (see Section 4.9.5).

 ©2006 BBN Technologies Corp.

4.8.6 Port Conventions

As discussed in Section 4.6.2, the *message* construct that is exchanged among *m-block*s and the metadata processing that is performed internally by *m-block*s provide a means to propagate control information through the a collection of *m-block*s, on a per unit of transport basis.

In principle, data, control and signaling information can all traverse the same path through a collection of *m-block*s. However, it may be desirable to use separate ports for control information, signaling information and data. The ports will use the same message passing and priority queueing mechanisms and will be subject to the same scheduler. However, they will transport different types of information. This capability shows the flexibility and power of the message passing port abstraction.

We propose to create the following standard protocol classes to transport information through collections of *m-block*s:

Command and Status Protocol Class. This protocol class will specify the types of command messages that can flow into an *m-block* and the type of status messagest that can flow out of that *m-block*. The command could be, for example, to initialize internal data structures, while the status could be a message to acknowledge that internal data structures had been initialized.

Data Path Protocol Class. This protocol class will specify the types of data messages that can flow into a *m-block*. In general, these flows will tend to be uni-directional, however, messages may flow in the reverse direction to provide support for reporting of status, etc.

Signaling Protocol Class. This protocol class will specify the types of signals that can flow into or out of a *m-block* for event notification. An event could be the arrival of a packet, or a notification from another entity that it is time to transmit a packet.

4.8.7 Finite State Machines

We propose to use hierarchical Finite State Machines (FSMs) to control the behavior of *m-block*sduring a given invocation of `handle_message`. Hierarchical FSMs provide the means to handle state complexity by allowing multiple sub-states to be composed in to a single higher level abstract state.

4.8.8 Miscellaneous Implementation considerations

In this section, we describe a few miscellaneous issues that may concern implementers.

Threading and reentrancy. A scheduler will *never* recusively invoke a given *m-block* instance. Thus, no concurrency control is required within any *m-block*. The underlying message sending primitive (and possibly other primitives) will be thread-safe, thus messages will be able to be sent between threads.

Inter-thread communication. As some degree of multi-threaded operation is likely, we will want utilities to support inter-thread signaling and inter-thread exchange of data.

Parallelization. We allow for the possibility of multi-threaded schedulers.

4.9 Abstract Syntax for GNU Radio Extensions

This section provides a semi-formal description *m-block*s, messages, protocol classes and how they relate. It borrows heavily from the formalism of Real-Time Object-Oriented Modeling[91].

©2006 BBN Technologies Corp.

4.9.1 Message

A *message* is a 5-tuple,

$$message =< signal, data, metadata, priority, rcvd\_port\_id > \tag{4.1}$$

Where *signal* is the identifier of the message and is often used for triggering finite state machine transitions. The *data* and *metadata* associated with a *message* are implemented as polymorphic types and represent the data to be operated upon and annotations of that data, respectively. The metadata object is typically an instance of a container that can implement a set of key / value mappings. The *priority* of the message is chosen from a small finite set and indicates to the underlying runtime system the urgency associated with this message. *rcvd_port_id* identifies which port the message arrived on, and may be used as an additional input when computing state transitions or other actions.

4.9.2 Message Type

The type of a message is defined by its *signal*

$$message\_type = signal \tag{4.2}$$

4.9.3 Protocol Class

A *protocol_class* is defined by a tuple,

$$protocol\_class =< class\_name, valid\_messages > \tag{4.3}$$

where *valid_messages* consists of two sets that specify the incoming and outgoing message types,

$$valid\_messages =< incoming\_message\_set, outgoing\_message\_set > \tag{4.4}$$

where each message set consists of zero or more message types,

$$message\_set = \{message\_type_1, message\_type_2, \ldots\} \tag{4.5}$$

4.9.4 *m-block* Type

In an abstract sense, an *m-block* has a type determined by its interface to its peers. The interface to its peers is a set of zero or more *port references* that appear on the outside of the *m-block*. See Figure 4.5.

$$m\_block\_type = peer\_interface \tag{4.6}$$

$$peer\_interface = \{port\_ref_1, port\_ref_2, \ldots\} \tag{4.7}$$

4.9.5 Port Reference

A *port_reference* is a 4-tuple,

$$port\_ref_i =< port\_ref\_name_i, replic\_factor_i, protocol\_class_i, conj\_ind_i > \tag{4.8}$$

$port\_ref\_name_i$ is the name of port reference i, and must be unique among all *port_refs* of the *m-block*. The replication factor, $replic\_factor_i$, is a tuple $< min, max >$ specifying the minimum and maximum number of connections that may be made to the port. *min* is a non-negative integer; *max* is an integer $\geq min$. The conjugation indicator, $conj\_ind_i$, is a boolean which if true indicates that the incoming and outgoing message sets of the protocol class associated with the port reference are swapped.

 ©2006 BBN Technologies Corp.

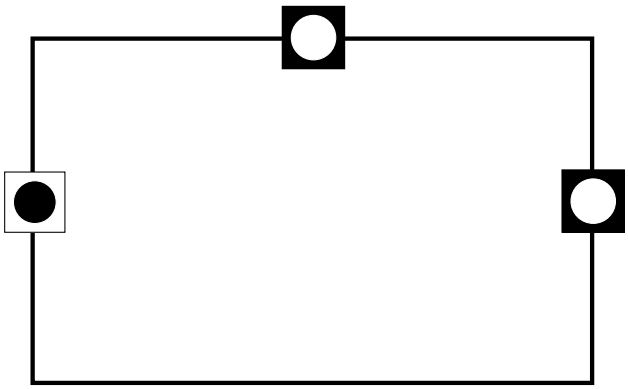

Figure 4.5: An *m-block* class with normal and conjugated external end ports

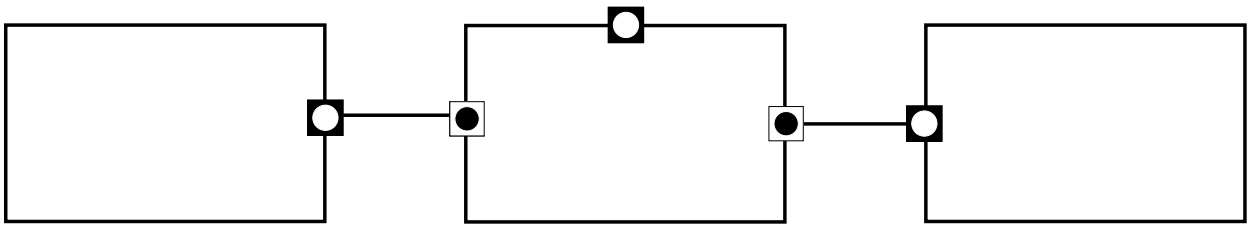

Figure 4.6: Compatible ports with connections

A port may only be connected to a compatible port. Two ports A and B are compatible if after applying any conjugation operators, the set of output messages of port A is a subset of the input messages of port B, and the set of output messages of port B is a subset of the input messages of port A. In the common case, both ports have the same protocol class, and one is conjugated. See Figure 4.6.

Ports with maximum replication factors greater than 1 are called *replicated ports*. By default, messages sent from replicated ports are copied and sent to all connected ports.

4.9.6 *m-block* Class

An *m-block* class is defined as a 3-tuple,

$$m\_block\_class =< class\_name, m\_block\_type, implementation > \tag{4.9}$$

4.9.7 *m-block* Implementation

The implementation of an *m-block* consists of two components,

$$implementation =< structure, behavior > \tag{4.10}$$

where *structure* determines the internal composition of the *m-block* and the *behavior* component specifies how it responds to received messages.

4.9.8 Behavior

This section does not discuss the implementation of an *m-block*'s behavior component beyond noting that:

- Messages are delivered to the behavior component by invoking its *handle-message* callback.

©2006 BBN Technologies Corp.

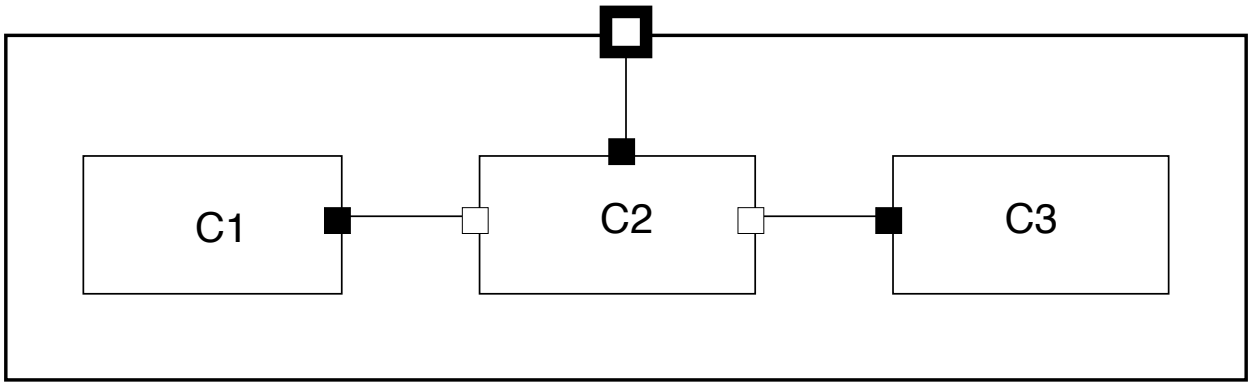

Figure 4.7: A composite *m-block* with relay port

- The behavior component may send messages to the ports that are directly accessible to it. Those ports are called *end ports*.
- The behavior component is typically realized as a finite state machine.

4.9.9 Structure

The structure component is defined as a 3-tuple,

$$structure =< end\_ports, components, connections > \tag{4.11}$$

end_ports is the set of ports that are directly accessible to the behavior component. In the case of a simple *m-block*(one that does not contain nested *m-block*s) *components* and *connections* are empty.

4.9.10 Components

components is defined as the set of references to *m-block*s contained in the structure component of an *m-block*,

$$components = \{m\_block\_ref_1, m\_block\_ref_2, \ldots\} \tag{4.12}$$

where each *m_block_ref* is defined by a 3-tuple,

$$m\_block\_ref_i =< m\_block\_ref\_name_i, replic\_factor_i, m\_block\_class_i > \tag{4.13}$$

$m\_block\_ref\_name_i$ names the reference and must be unique among all *m_block_refs* of the *m-block*. $replic\_factor_i$ is a tuple $< min, max >$ specifying the minimum and maximum replications of the given *m_block_ref*. *min* is a non-negative integer; *max* is an integer $\geq min$. The replication factor is typically $< 1, 1 >$. $< 0, 1 >$ indicates that the *m-block* reference is optional.

4.9.11 Classification of Ports

There are three different kinds of ports. See Figure 4.8:

Relay ports are part of the peer interface of the *m-block* and may be connected to a component *m-block*. Messages coming through a relay port are *not* seen by the behavior component. Messages arriving at an unconnected relay port are discarded. See Figure 4.8, port **p3**.

External end ports are part of the peer interface of the *m-block* and are not relay ports. These are directly accessible by the behavior component. See Figure 4.8, ports **p1** and **p2**.

 ©2006 BBN Technologies Corp.

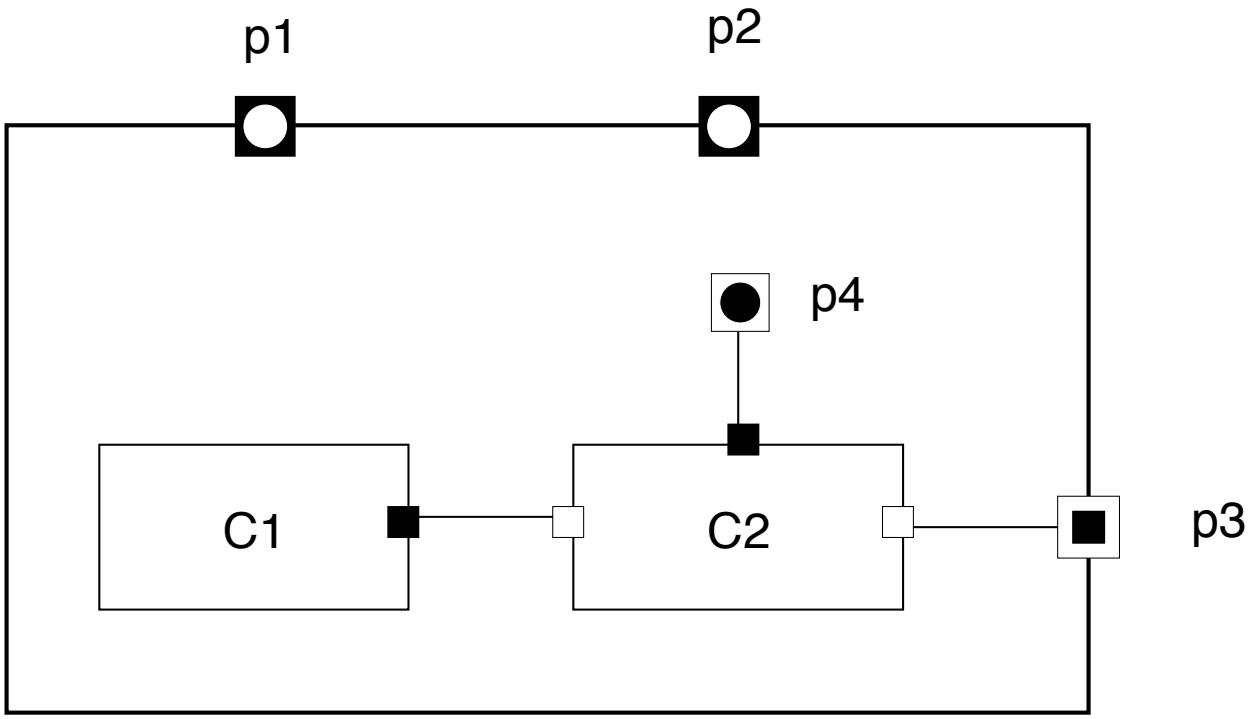

Figure 4.8: A composite *m-block* illustrating all port types

Internal end ports are ports used to connect the behavior component to an internal component *m-block*. These ports are not visible from the outside of the *m-block*, and are not a part of the peer interface. See Figure 4.8, port **p4**.

The following relationships hold:

$$relay\_ports \subseteq peer\_interface \tag{4.14}$$

$$external\_end\_ports \subseteq peer\_interface \tag{4.15}$$

$$external\_end\_ports = peer\_interface \setminus relay\_ports \tag{4.16}$$

$$internal\_end\_ports \not\subset peer\_interface \tag{4.17}$$

$$end\_ports = external\_end\_ports \cup internal\_end\_ports \tag{4.18}$$

$$ports = end\_ports \cup relay\_ports \tag{4.19}$$

4.9.12 Connections

Connections between *m-block*s contained in the structure component of an *m-block*s are defined by the possibly empty set,

$$connections = \{connection_1, connection_2, \ldots\} \tag{4.20}$$

Where,

$$connection_i =< end\_point_1, end\_point_2 > \tag{4.21}$$

and the end points of the connection are specified by

$$end\_point_i =< m\_block\_id_m, port\_id_n >, i = 1, 2 \tag{4.22}$$

©2006 BBN Technologies Corp.

which can specify either a port reference on the *m-block* itself,

$$m\_block\_id_m = self \tag{4.23}$$

$$port\_id_n \in ports \tag{4.24}$$

or a port reference to a peer interface port of a contained *m-block*,

$$m\_block\_id_m \in components \tag{4.25}$$

$$port\_id_n = peer\_interface\_name(n, class(m\_block\_id_m)) \tag{4.26}$$

4.10 Walk through Example: Interaction with MAC and PHY Layers

Figure 4.9 shows a collection of *m-block*s connected into a graph that would support a typical MAC-layer (such as 802.11). A single USRP is at the bottom of the graph which has two branchs: a transmit branch and a receive branch. In this example, we consider the case where the receive side of the graph receives a Clear To Send (CTS) message indicating that the node has the channel and should transmit as soon as possible.

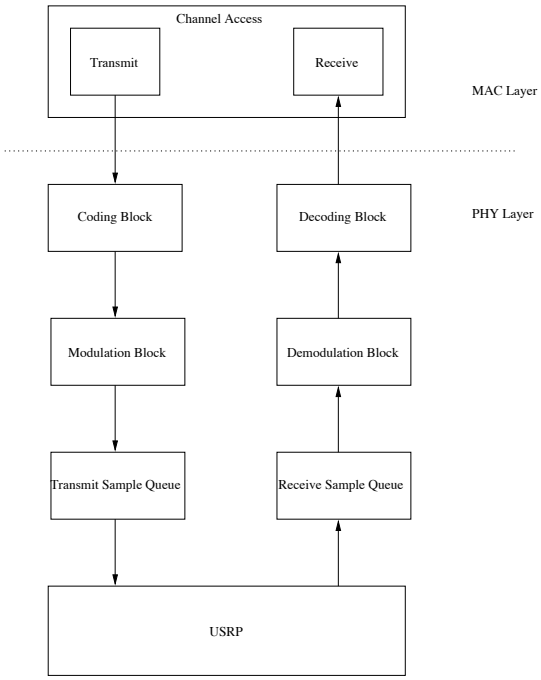

Figure 4.9: MAC/PHY Integration Example

The Transmit Sample Queue contains the raw RF samples for the next packet to be transmitted. These samples can be fed directly into the USRP for transmission at the appopriate time. The Transmit Sample Queue blocks waiting for a high priority signal to come to indicate that it is time to transmit. In the meantime, processing can go on in other parts of the graph (another packet could be processed higher up on the transmit side of the graph or on the receive side). When the CTS is decoded, a high priority signal is generated. In normal 802.11 implementations using dedicated processing hardware, the CTS message is

 ©2006 BBN Technologies Corp.

propagated up to the MAC layer which is then able to trigger the next packet transmission. This processing is usually implemented in hardware and so the time from when a CTS message is received to the time the next message is transmitted can be as short as tens of microseconds. However, This rapid turnaround time may not be possible on some platforms (for example a general purpose PC that has to contend with various bus latencies in order to transfer data between its components) even though the scheduling mechanism supports it. For these platforms, the signal may be sent directly form the Decode Block to the Transmit Sample Queue, triggering the transmission of the next packet to the USRP and then over the air. For some platforms, even this optimization may not guarantee a response that is rapid enough to meet tight timing requirements. Specialized processing, for example using specialized FPGA structures, might be needed. The extensions we propose assumes that less stringent timing requirements will be adequate in the short term, and that we will revisit alternatives to improve timing predictability and latency in a later version, after the current approach has been evaluated on current target platforms with more simply structured MAC's that have less timing sensitivity.

4.11 Summary

We propose a number of extensions to the GNU Radio baseline architecture that we believe will provide improved support for packet radio protocols. The changes we propose can be implemented in an incremental fashion as separate modules on the existing GNU Radio framework and will not impact existing software or functionality.

The main extension proposed is a new *message*-block or *m-block* that is capable of message processing streams that are composed of data and metadata. This new block is capable of processing timing information and maintaining timing synchronization between other *m-block*s, and exposes a control plane, where *m-block*s may exchange event signals and commands between each other as well as with attached programs operating in other process contexts, and even on other processors or physical hosts. The *m-block*s integrate with a proposed top-level scheduler to support hierarchical, quasi-real-time scheduling. The *m-block*s also integrate readily with existing GNU Radio blocks and scheduling algorithms.

In addition to the new block type, we propose the addition of the following elements/capabilities:

- Message-based Scheduling.
- Combined data and metadata.
- Standardized time transfer mechanism.
- Dynamic re-configurability.
- Support for control interfaces.
- Hierarchical, quasi-real-time, hybrid scheduler.
- Memory Management to support the functionality above.

These proposed features would be implemented in close collaboration with the GNU Radio community. It is our hope and desire that the extensions to be made will be useful to the global GNU Radio community. We are grateful for any comments and/or suggestions from this community and look forward to an enjoyable and fruitful collaboration.

©2006 BBN Technologies Corp.