Software interfaces to protocol plug-ins

Draft 4.4 (2011 October 19)

Concepts

RCE code is divided into components:

1. The core, which is the same for all RCEs of a given generation. It contains low-level processor management code, RTEMS, generic C/C++ support libraries, etc.
2. The protocol plug-in (PPI) software modules.
3. Application code which is entered only after the first two components are fully initialized.

Each component's code is normally stored independently somewhere on the RCE, e.g., in configuration flash.

The core and the PPI software modules together offer the use of protocol plugins at different levels of abstraction:

1. Low-level for RCE startup code and for plugin software modules.
2. Mid-level for raw device users.
3. High-level for most applications.

This document describes the first two levels of PPI service interface which uses abstractions called ports, virtual channels, payloads (data), conduits and factories.

Mid-level

A port represents a hardware I/O engine capable of DMA to and from system RAM, i.e., one firmware instance of a protocol plug-in of a particular type. An RCE has at most eight ports (which limit derives from the limit on the number of plugins). Each type of port has both an official type number and an official short name such as "eth" or "config". Ethernet ports that differ only in speed, e.g., 10 Gb/s ethernet and a 100 Mb/s ethernet have the same port type and short name. Each port has a certain number of virtual channels which may be allocated and deallocated upon request from the user; some types of ports may have as few as one virtual channel while others may have thousands.

A virtual channel is one end of a two-way communications link similar in concept to a BSD socket. The VCs are globally visible but not MT-safe; at a given time at most one thread may be making use of a given virtual channel. Each virtual channel represents a different source and/or sink of data such as a UDP port or a section of Petacache memory. Each has a number that uniquely identifies it amongst all the virtual channels belonging to the same port. When asking for a virtual channel to be allocated the user may specify a specific number that may have meaning at a higher level, e.g., it may map to a UDP port number. Even when asking for a specific virtual channel number no channel may be allocated more than once. The other method of allocation just picks some virtual channel that has not yet been allocated. When finished with a virtual channel the user passes it back to the owning port for deallocation.

To transmit an outgoing message the user requests an the address of an empty message buffer from the virtual channel. After filling this the user then passes buffer address back to the virtual channel for transmission. To receive an incoming message the user makes a request that blocks until a message is available, uses the data at the buffer address eventually supplied then returns the buffer to the virtual channel that supplied it.

At this level each buffer contains the payload of a single inbound or outbound message where the boundaries between messages are respected; in other words the virtual channel implements datagram rather than byte-stream protocols. It's up to high level software such as an IP stack to provide any operations that cross message boundaries. Any message headers or other system overhead are managed by the low-level software.

Low-level

Each message coming in on a port must contain information from which a destination virtual channel can be inferred. If it doesn't, or if it specifies a virtual channel that is invalid or not allocated, then the port's lost-message count is incremented and the message is discarded (its buffer being reclaimed).

A port usually connects to pins leading out of the FPGA on which the RCE resides, though some ports may simply offer access to local resources such as DSPs. No pin may be used by more than one port.

The FPGA is connected to the larger system by data paths called conduits. Each conduit connects to a group of pins on the FPGA; no pin may belong to more than one conduit. At system startup each port that uses pins will be matched to the conduit that connects to exactly the same set of pins. If the resulting mapping is not one-to-one then the startup fails. Each conduit has an associated type number and version number which the software for the matching port checks for validity at boot time. If the software rejects the conduit then again startup will fail.

The lowest level of plugin software is held in relocatable modules recorded on the RCE. A module for plugin type FOO holds:

- An implementation of a class FooPort derived from the abstract class Port.
- An implementation of class FooFactory derived from the abstract class Factory. A FooFactory creates FooPort instances which are returned as `Port*` values.
- An entry point that creates an instance of FooFactory which is returned as a `Factory*` value.

Each module is loaded, relocated and bound to the system core before its first use. The core code knows only the abstract base classes Factory and Port, not the derived classes specific to plugin type.

Both plugin hardware and the port-factory modules have version numbers which will allow some measure of compatibility checking at boot time. Any incompatibility detected causes startup to fail.

RCE startup code discovers the set of protocol plugins and conduits available using a set of configuration registers. These registers have their own address space, the "configuration space". For each plugin the configuration space registers yield the plugin type, the plugin version number and the set of FPGA pins connected to it. For each conduit they yield the conduit type, the conduit version number and its set of FPGA pins.

Access to configuration registers is via an object which given an abstract register address reads or writes the contents of the corresponding register. Another object uses the abstract register layer to provide all the configuration info for a plugin or a conduit given its index number (or supplies an indication that the given entity doesn't exist).

Differences between Gen I and Gen II

Gen II

The configuration space registers are hardware memory locations filled with information by the IPMI Controller (IPMC).

Message buffers, once created, are managed entirely by firmware. Payload and headers are managed independently of each other. The firmware manages buffers directly using the buffer addresses. The size of the buffer required for (the payload part of) a message is requested separately for each message.

Gen I

There is no IPMC and there are no hardware configuration registers. The low-level configuration information is burned into a configuration flash container. At boot time the container is read and its contents are used to construct an object which simulates the Gen II configuration register space. Above that layer the handling of the information is just like that in Gen II.

Protocol plugins are assembled from Protocol Interface Core (PIC) blocks which have no counterpart in Gen II. We want to make the configuration information handling like that in Gen II, so rather than extending it with PIC block assignments we sweep those assigments under the rug by embedding them in the plugin software modules.

Control of I/O buffers passes between firmware and application software; a Transfer Descriptor Entry (TDE) is used to pass buffer references in either direction. Frame headers are fully exposed to software which must know for a given plugin the header size, the maximum payload size and the maximum number of buffers for a given plugin. This information can also be embedded in the plugin-handling module at the cost of having to tailor the module to the RCE application. All the buffers for a given plugin (or even a group of related plugins) are of the same size and are preallocated before any messages are sent.

Mid-level API

The class declarations given in this section contain only those members intended for use after system startup is complete and all plugins are on-line. Whether a method is virtual is not specified, nor are friend declarations shown; these are considered implementation details.

Classes and their responsibilities

Class name	Instance responsibilities
Port	Represent a single protocol plugin. Allocate and deallocate virtual channels. Deliver data to and from virtual channels. Retain the configuration information for the plugin and the index number of the conduit (if any) assigned to it at startup. Print multi-line reports on the plugin state and configuration.
PortList	Keep a linked list of all Port instances. Assign each an global index number not used by any other Port. Assign each a second index number not used by any Port of the same type. Search the list by global index number, by type and type index number or by conduit number. Print a brief report on the status of all ports, one line per port.
VirtualChannel	Represent a single virtual channel associated with the allocating Port. Accept message payloads for transmission. Return messages that have been received (waiting for them if needed).

Universal constants (`constants.hh`)

All RCEs whether of Gen I or Gen II each have the same limits on the number of plugin instances (`MAX_PLUGINS`).

```
static const unsigned MAX_PLUGINS  =  8;
```

Port-type enumeration (`PortTypes.hh`)

The numbers are members of an enumeration assigned by the Data Acquisition Tools (DAT) project.

```
enum PortType {
    CONFIG_FLASH,
    ETHERNET,
    PGP,
    etc.,
    INVALID_PORT_TYPE
};
```

The header file also contains a specialization of the template `tool::type::EnumInfo` which allows one to use the function templates `emin<>()`, `emax<>()`, `ecount<>()`, `evalid<>()`, `enext<>()`, `eprev<>()` and `estr<>()`:

```
emin<PortType>() == CONFIG_FLASH
emax<PortType>() == PortType(INVALID_PORTTYPE - 1)
ecount<PortType>() == int(INVALID_PORTTYPE)
evalid(PortType x) is true for all from emin() to emax() inclusive, else false
evalid(int) and evalid(unsigned) make similar tests on ints and unsigneds.
enext(emax()) == eprev(emin()) == INVALID_PORTTYPE
enext(CONFIG_FLASH) == ETHERNET, etc.
eprev(ETHERNET) == CONFIG_FLASH, etc.
estr(CONFIG_FLASH) == "CONFIG_FLASH", etc.
estr(x) == "**INVALID**" if and only if evalid(x) is false
```

`ecount<>()` can't be used as a dimension for static arrays since the compiler considers it to be non-constant; in that case use `EnumInfo<PortType>::count`.

Port list (`PortList.hh`)

This class is a Borg-type singleton; the constructor makes a stateless object whose member functions access the true (shared) state defined elsewhere. The shared state is constructed at system startup. The destructor destroys these stateless objects but does not touch the true state information. You can therefore just use the constructor whenever you need to access the One True List, e.g., `PortList().head()`.

You can get a count of the number of ports or the first port on the list (the list can't be empty). The `report()` member function will print informational messages in the system log which show the contents of the port list in brief form, one line per port.

> ⚠ The location and form of the system log depends on how the system logging package was initialized at application startup. Client code making log entries is not aware of this initialization.

A particular port may be looked up in several different ways:

- By its global index number, assigned in sequence starting from zero as ports are created.
- By its type number and the index number within the type. The first ethernet port would be `(ETHERNET,0)`, the second `(ETHERNET,1)`, etc.
- By the number of the conduit the port is connected to.

Lookup methods return the null pointer if the search fails.

```
class PortList {
public:
    PortList() {}
    ~PortList() {}
    int numPorts() const;
    Port* head() const;
    Port* lookup(PortType type, unsigned typeIndex) const;
    Port* lookup(unsigned index) const;
    Port* lookupByConduit(unsigned conduit) const;
    void report() const;
);
```

Port (`Port.hh`)

A port object represents a particular instance of a protocol plug-in. Each port object is created at system startup. Port objects live until system shutdown and may not be copied or assigned.

Each port allocates and deallocates VirtualChannel objects on demand. During its lifetime each VirtualChannel object has exclusive use of one of the port's virtual channel numbers; the virtual channel number becomes available again once the VirtualChannel object is deallocated. The application code may request a specific, unused virtual channel number for the type of port, e.g., a well-known TCP port number. The application may also allow the port to assign a number not currently in use by any VirtualChannel.

Every port object is a member of the linked list accessed though class PortList and may not be removed from the list. Use the `next()` member function to iterate over the list.

A short name for the type and a short description of the port are also provided.

A "lost" counter is provided which counts the number of inbound messages that were discarded, for whatever reason.

Other information provided:

- The index number of the conduit associated with the port.
- The hardware version number of the associated plugin.
- The software version number of the associated plugin module.
- A bitmask giving the FPGA pins (if any) used by the plugin.
- The size of the VC-number space.

The `report()` member function produces detailed multi-line description of the port in the system log, including all platform-specific information.

High-level and mid-level code isn't allowed to create or destroy instances; only low-level code is allowed to do that.

```
class Port {
public:
    VirtualChannel* allocate(int vcNum);
    VirtualChannel* allocate();
    void deallocate(VirtualChannel *);
    unsigned lost() const;
    unsigned index() const;
    unsigned type() const;
    const char* name() const;
    unsigned typeIndex() const;
    uunsigned conduit() const;
    unsigned versionHard() const;
    unsigned versionSoft() const;
    Port* next() const;
    const char* description() const;
    unsigned maxVcs() const;
    void report() const;
};
```

Virtual channel (`VirtualChannel.hh`)

Each VirtualChannel object is allocated by a Port and is assigned a unique ID in the Port's virtual channel number space.

Messages inbound on the associated port may be waited for and retrieved using the `receive()` member function, which returns a `void*` pointer to the payload portion of a message. Client code will normally keep a payload for a short time then give it back to the virtual channel they got it from using the virtual channel's `deallocate()` member function. It's an error to request a virtual channel to deallocate a payload it didn't produce; the result of doing so will be unpredictable.

A virtual channel takes message payloads given to its `transmit()` member function via `void*` pointers and queues them for output. The payload pointer must have been produced by the `allocate()` member function of the same virtual channel (or its `receive()`); breaking this rule results in unpredictable behavior. Once the message is transmitted the message buffer is automatically deallocated, so the user should not try to use it after calling `transmit()`. When allocating a buffer the client must specify the maximum size of the payload to be transmitted (in Gen I systems this is ignored since all buffers for a port will be the same size).

```
class VirtualChannel {
public:
    unsigned vcNum() const;

    void* receive();          // To receive: first call this ...
    void deallocate(void*);   // ... then this.

    void* allocate(size_t payloadSize); // To transmit: first call this (or receive()) ...
    void transmit(void*);               // ... then this.
};
```

Low level API

In this section we describe the code used to manage configuration information and construct the global PortList. Some of the classes already introduced above will have new members described here; other classes will be completely new.

New members of old classes

The PortList class has a static member function `build()` whose main purpose is to produce the list of Port instances. To do so it will have to read configuration information about plugins and conduits, load and activate plugin software and match conduits to ports. There is also a member function `add()` which places a new Port on the end of the list.

```
class PortList {
  public:
  static void build();
  private:
  void add(Port*);
};
```

The Port class' constructor is used by `PortList::build()`. It's constructor and destructor are used by derived classes. Also provided are the means to increment the count of lost messages, to set the next-port member and to find out the set of FPGA pins used by the port.

```
class Port {
protected:
  Port(unsigned index,
       PortType type,
       const char *name,
       unsigned typeIndex,
       unsigned conduit,
       unsigned versionHard,
       unsigned versionSoft,
       const char* description,
       unsigned long long pins,
       unsigned maxVcs);
   virtual ~Port() = 0;
   unsigned long long pins() const;
   void incLost();
   void next(Port*);
};
```

Instances of VirtualChannel are created and destroyed only inside Ports. There is an access member added which gives the owning Port instance.

```
class VirtualChannel {
private:
  VirtualChannel(Port*, unsigned vcNum);
  ~VirtualChannel();
  Port* port() const;
};
```

New classes and their responsibilities

Class name	Instance responsibilities
ConduitConfig	Hold the configuration information for one conduit.
ConfigReader	Collect all the available information about a given plugin (conduit) from ConfigSpace and put it into an instance of PluginConfig (ConduitConfig). Indicate when the given plugin or conduit doesn't exist.
ConfigSpace	Provide an address space of abstract 32-bit registers containing configuration info for plugins and conduits, whether or not such registers exist in hardware.
PluginConfig	Hold the configuration information for one plugin instance.
PortFactoryList	Hold all PortFactory objects created during system startup. Look up factory instances by type.

Class /enum name	Class/enum responsibilities
ConduitType	Enumerate the different types of conduit.
PortFactory	Abstract base class for objects that given an instance of PluginConfig and an instance of ConduitConfig produce a Port instance. The ConduitConfig is optional for plugins that don't connect to a conduit.

ConduitConfig (`ConduitConfig.hh`)

This class describes a single conduit.

Member	Description
index	The order of appearance, starting from zero, of the information in ConfigSpace.
type	The type of conduit.
version	The version number of the conduit definition.
pins	Has a 1 bit for each FPGA pin connected to the conduit.

The default constructor creates an invalid instance, one that represents a conduit that doesn't exist. A member function tests whether the instance is a valid one.

```
class ConduitConfig {
public:
  unsigned index;
  ConduitType type;
  unsigned version;
  unsigned long long pins;
  ConduitConfig();
  ConduitConfig(unsigned ind, ConduitType, unsigned ver, unsigned long long);
  bool isValid() const;
};
```

ConduitType (`ConduitType.hh`)

These types are not well defined yet so for now we just define a generic type code. The header also provides a specialization of `tool::type::EnumInfo<>` similar to that provided for PortType.

```
enum ConduitType {
  CONDUIT,
  INVALID_CONDUIT_TYPE
};
```

ConfigReader (`ConfigReader.hh`)

One member function returns instances of PluginConfig, the other returns instances of ConduitConfig. Both take an argument that is the index of the object whose configuration you want to look up; plugins and conduits are numbered separately starting from zero. The lookups return invalid config instances if the requested entities don't exist.

Instances have no data of their own but get what they need from ConfigSpace; you can generate and throw away instances as often as you want.

```
class ConfigReader {
  public:
  void lookupConduit(unsigned index, ConduitConfig&) const;
  void lookupPlugin(unsigned index, PluginConfig&) const;
};
```

ConfigSpace (`ConfigSpace.hh`)

Each instance implements an abstract space of configuration registers. How the abstract registers are used to collect configuration information is an implementation decision which will however be the same for both Gen I and II. Register addresses start at zero; an attempt to read or write a register at an invalid address, or to write to a read-only register, will throw `std::logic_error`. Use the `implements()` member function to determine if a virtual register is implemented at a given address.

Instances have no data of their own but get what they need from some central source on the RCE; exactly where differs between Gen I and Gen II. You can create and destroy instances at will.

```
class ConfigSpace {
  public:
  ConfigSpace();
  bool implements(unsigned address) const;
  unsigned read(unsigned address) const;
  void write(unsigned address, unsigned value);
};
```

PluginConfig (`PluginConfig.hh`)

This class describes a single plugin instance.

Member	Description
index	The global index number of the plugin.
type	The type of Port to make for the plugin.
version	The version number of the plugin definition.
pins	Has a 1 bit for each FPGA pin connected to the plugin.

The default constructor creates an invalid instance, one that represents a plugin that doesn't exist. A member function tests whether the instance is a valid one.

```
struct PluginConfig {
  unsigned index;
  PortType type;
  unsigned version;
  unsigned long long pins;
  PluginConfig();
  PluginConfig(unsigned ind, PortType, unsigned ver, unsigned long long);
  bool isValid() const;
};
```

PortFactory (`PortFactory.hh`)

This is an abstract base class. Once the system startup code knows the types of the available plugins it will load the plugin software module for each type. It will call the entry point of each plugin software module once to obtain an instance of a class derived from PortFactory.

Once it has matched a PluginConfig instance with a ConduitConfig instance, or determines that the plugin needs no conduit, the startup code uses the factory object to create Port instances for the given type of plugin. If the plugin and conduit versions are incompatible the factory member function will throw `std::logic_error`. It will do the same if no ConduitConfig is supplied when one is required.

The report function will log full details of the factory, including any platform-dependent information.

```
class PortFactory {
public:
  PortType type() const;
  const char* name() const;
  unsigned version() const;
  const char* description() const;
  Port* makePort(const PluginConfig&);
  Port* makePort(const PluginConfig&, const ConduitConfig&);
  PortFactory* next() const;
  void next(PortFactory*);
  void report() const = 0;
protected:
  PortFactory(PortType, const char* name, unsigned version, const char* description);
  ~PortFactory();
};
```

PortFactoryList (`PortFactoryList.hh`)

Another Borg singleton, very similar in concept to PortList. This list of factories is built at about the same time as the list of ports. There is at most one factory per port type. The lookup function returns a null pointer if no matching factory is on the list. The report function logs a one-line summary per factory. New PortFactory instances are added to the end of the list.

```
class PortFactoryList {
public:
  PortFactoryList();
  PortFactory* head() const;
  unsigned numFactories() const;
  PortFactory* lookup(PortType) const;
  void report() const;
  void add(PortFactory*);
};
```

Plugin software module interface `PluginModule.hh`

Each module's entry point is named `rce_appmain`; this symbol is recognized by the module building system which places its value in the transfer address slot of the module's ELF header. The prototype of the entry point function is

```
extern "C" PortFactory* rce_appmain();
```

The system startup code uses class PluginModule to find existing plugin modules and run them, or to save plugin software modules in the usual place given their images in memory. The first constructor fetches the module from some internal RCE storage while the second one reads it from a file. In either case one may then write the module to internal storage or run it to obtain the factory object. Once the module has been run any attempt to write it will throw `std::logic_error` because the module code will no longer be relocatable.

```
class PluginModule {
public:
  typedef PortFactory* (*EntryPoint)();
  explicit PluginModule(PortType);
  PluginModule(PortType, const char* filename);
  PortFactory* run();
  void write() const;
};
```

The module's entry point is run directly without creating a new thread (otherwise we'd need synchronization in order to wait for the factory to be produced).

Gen I-specific initialization

On Gen I RCEs the application software is left with the job of allocating I/O buffers and it can't do that without knowing for each plugin the header sizes, max payload sizes and max number of message buffers for import and export. That information is available from the port factories but it means downcasting the `PortFactory*` values gotten from the PortFactoryList. TDEs for the inbound buffers must be pushed into one or more FLBs before any data may be received. The interface described here lets the application do the needed initialization without exposing the innards of the plugin-handling system.

Ethernet (TBD)

PGP

PgpSetup (`PgpSetup.hh`)

Creating an instance of this class performs the following functions:

- Allocates I/O buffers.
- Write TDEs to the FLB FIFO.
- Brings up the MGT links for the requested ports.
- Resets the appropriate PIC blocks and enables their events.

If you request a PGP port that does not exist then the constructor will throw `std::logic_error`. If any other initialization fails the constructor will throw s `td::runtime_error`.

Early versions will used cached memory for I/O buffers as has been done in the past. Later versions will used uncached memory. Buffers for inbound messages are shared amongst all PGP ports while each port has its own pool of buffers for outbound messages. The default is to to use all PGP ports and to allocate the maximum number of each kind of buffer with the maximum payload size allowed for the PGP ports. If you request more buffers than the maximum then the maximum number is allocated.

The destructor deallocates all the buffers allocated by the constructor and resets the selected PIC blocks again, this time disabling their events.

Though header sizes and max payload sizes for inbound and outbound messages differ slightly, this class will allocate the maximum for each buffer. Inbound and outbound buffers will differ only in how the length parameter is set in the transaction descriptor since its interpretation varies depending on the direction of transfer.

```
class PgpSetup {
public:
  enum Port {PORT0=1, PORT1=2, PORT2=4, PORT3=8};
  enum {ALL=-1, DEFAULT=-1};

  explicit SetupPgp
  (int portMask                 =ALL,  // Either ALL or a logical OR of values from enum Port.
   int numInboundBuffers        =ALL,
   int numOutboundBuffersPerPort=ALL,
   int maxPayloadSize           =ALL,
   int resumeThreshold          =DEFAULT  // Either DEFAULT or the FLB resume threshold.
  );

  ~SetupPgp();

  int portMask()                  const;
  int numInboundBuffers()         const;
  int numOutboundBuffersPerPort() const;
  int headerSize()                const;
  int maxPayloadSize()            const;
  int resumeThreshold()           const;
};
```

Gen I-specific low-level API

Buffer (`Buffer.hh`)

On Gen I hardware each message is represented by a data structure in main memory called a Transaction Descriptor. The descriptor contains pointers to all the other message-related data:

- Message header
- Message payload
- Transaction completion descriptor.

The allocation and preparation of the descriptor and the other message data is the responsibility of software; the firmware doesn't manage them. For simplicity each message is represented in software by an instance of class Buffer. Each Buffer contains a Transaction Descriptor, Transaction Completion Descriptor, header buffer, payload buffer and next/previous Buffer pointers all welded together into a single object. Each part will have a fixed offset which is hard-coded into the object so that we don't have to refer to non-cached memory just to find out their addresses. For that reason we allocate a fixed number of bytes for the header no matter the type of plugin; the largest header is 32 bytes for the Petacache PGP non-register messages so we'll allocate twice that. The Transaction Descriptor has the strictest alignment requirement (see below) so it comes first. The payload is of variable size so it comes last.

The first word of a Transaction Descriptor is a length parameter whose interpretation depends on the direction of transfer. For outgoing messages it's the number of payload bytes to send. For incoming messages it's the maximum number of header+payload bytes that will be accepted. The plugin never alters the Transaction Descriptor so to see how many bytes were transferred one has to examine the transfer count in the Transaction Completion Descriptor. For reception the completion descriptor is always updated by the plugin while for transmission this happens only when the transaction fails.

A PIC block gives or takes a reference to a Transaction Descriptor in the form of a 32-bit value called a Transaction Descriptor Entry (TDE). Six of those bits are reserved for various flags, so a TDE contains only the upper 26 bits of the address of the descriptor. Perforce the descriptor must be allocated on a 64-byte boundary. The completion descriptor must be allocated on a PowerPC cache line boundary (32-byte boundary). Due to a design quirk PIC blocks write **two** complete cache lines when updating a completion descriptor, so the space allocated to each must be artificially enlarged.

For details about the descriptors see chapter 4 of the Cluster Element Module document at http://www.slac.stanford.edu/exp/npa/design/CEM.pdf

⚠ Each instance is allocated inside a message buffer using placement new so that the member "m_payload" overlaps the first byte of the payload area. Early versions of the plugin software will allocate Buffers in cached memory as has been done in the past; later versions will use non-cached memory.

```
#include <rtems.h>

#define TRANSACTION_DESCRIPTOR_ALIGNMENT (64)
#define DMA_ALIGNMENT (PPC_CACHE_ALIGNMENT)

struct Buffer {

  enum {
    MAX_HEADER_SIZE  = 64,
    COMPLETION_DESCRIPTOR_SIZE  = 2 * PPC_CACHE_ALIGNMENT
  };

  struct Completion {
    unsigned parameter: 24;
    unsigned wasBlockError: 1;
    unsigned reason: 6;
    unsigned wasError: 1;
    uint32_t transferCount;
  };

  struct Transaction {
    uint32_t    lengthParameter;
    void*       const headerPtr;
    void*       const payloadPtr;
    volatile Completion* const completionPtr;
    Transaction(void* hPtr, void* pPtr, volatile Completion* cPtr);
  };


private:
  Transaction         m_transaction              __attribute__((aligned(TRANSACTION_DESCRIPTOR_ALIGNMENT)));
  Buffer*             m_next;
  Buffer*             m_prev;
  union {
    volatile Completion m_completion             __attribute__((aligned(DMA_ALIGNMENT)));
    uint8_t           m_pad0[COMPLETION_DESCRIPTOR_SIZE];
  } m_paddedComp;
  mutable uint8_t     m_header[MAX_HEADER_SIZE]   __attribute__((aligned(DMA_ALIGNMENT)));
  mutable uint8_t     m_payload;                  __attribute__((aligned(DMA_ALIGNMENT)));

public:
  static Buffer*      transactionToBuffer(Transaction*);
  static Transaction* bufferToTransaction(Buffer*);
  static Buffer*      tdrToBuffer(unsigned tdr);
  static unsigned     bufferToTdr(Buffer*);
  static Buffer*      payloadToBuffer(void*);

public:
  Buffer();
  Buffer*                    next()           const;
  Buffer*                    prev()           const;
  const volatile Completion* completion()     const;
  void*                      header()         const;
  void*                      payload()        const;
  void                       next(Buffer* p);
  void                       prev(Buffer* p);
  unsigned                   lengthParameter() const;
  void                       lengthParameter(unsigned);
};
```

Platform-specific information (`PlatformInfo.hh` and `PlatformPluginInfo.hh`)

The single `PlatformInfo` instance is basically an array of `PlatformPluginInfo` instances. The array holds information that has no place in the `PortList` instance and is searched in a similar manner. The factory classes for Gen I hardware will fill in the information inside their `makePort()` member functions. Later on such classes as `PgpSetup` will retrieve it.

The constructor returns a reference to the single instance. {addPlugin()}} adds a new plugin to the array, fills in the port type and index within type, returning a pointer to the new entry. The `lookup()` member functions behave much like their counterparts in class `PortList`.

```
class PlatformInfo {
public:
  PlatformInfo() {}
  const char*         platform    ()                                        const {return "ppc405-rtems";}
  PlatformPluginInfo* addPlugin   (ppi::basic::PortType, unsigned typeIndex);
  PlatformPluginInfo* lookupPlugin(ppi::basic::PortType, unsigned typeIndex);
  PlatformPluginInfo* lookupPlugin(unsigned index);
};
```

Each entry in the array is a simple struct containing the required information such as header sizes, payload sizes and PIC block assignments for the given port. The limit on import buffers is understood to apply to the FLB assigned to the port rather than the port itself. This is important when multiple ports share a common FLB as is done with PGP.

```
struct PlatformPluginInfo {
  basic::PortType type;
  unsigned index;
  unsigned typeIndex;
  unsigned importHeaderSize;
  unsigned maxImportPayloadSize;
  unsigned maxImportBuffers;
  unsigned exportHeaderSize;
  unsigned maxExportPayloadSize;
  unsigned maxExportBuffers;
  unsigned pibNum;
  unsigned pebNum;
  unsigned flbNum;
  unsigned ecbNum;
};
```

Ethernet (TBD)

PGP

Petacache

Port factory class (`PetacachePgpFactory.hh`)

An instance of this class is what the PGP plugin module will use to create instances of `PgpPort`. The first `makePort()` member function is non-functional and will throw `std::runtime_error` if called. The `report()` member function will log some messages describing the platform-dependent features of the PGP implementation.

```
class PetacachePgpFactory: public basic::PortFactory {
public:

  PetacachePgpFactory();

  virtual ~PetacachePgpFactory();

  virtual PgpPort* makePort(unsigned index, unsigned typeIndex, const basic::PluginConfig&);

  virtual PgpPort* makePort(
    unsigned index,
    unsigned typeIndex,
    const basic::PluginConfig& plugin,
    const basic::ConduitConfig& conduit);

  virtual void report() const;

};
```

Gen I PGP plugins all share the same ECB and FLB but each is assigned its own PEB and PIB. The index you pass to the accessor function is the index within the plugin type PGP, the same number you would obtain from `Port::typeIndex()`.

Gen I initialization of ConfigSpace

Configuration container zero in the configuration flash contains tables of information about the hardware and firmware; this information can't be gotten directly from the hardware and firmware.

Not knowing the actual layout of the RCE's circuit board(s) I've arbitrarily assigned conduit 0 to the 10 Gb ethernet and conduits 1, 2, 3 and (for petacache) 4 to plugin type PGP. The four MGT's assigned to the ethernet I assign to pins 0-3 while the ones for PGP I assign to pins 4, 5, 6 and (peta) 7.

The container contents consist of eight instances of PluginConfig followed immediatly by eight instances of ConduitConfig, except that the "index" members are not recorded. For a Petacache RCE board (PGP only) the tables look like this: (substitute 0xffffffff for "EMPTY" and):

Type	Version	Pins
PGP	1	0x00000000 00000010
PGP	1	0x00000000 00000020
PGP	1	0x00000000 00000040
PGP	1	0x00000000 00000080
EMPTY	0	0x00000000 00000000
EMPTY	0	0x00000000 00000000
EMPTY	0	0x00000000 00000000
EMPTY	0	0x00000000 00000000

Type	Version	Pins
CONDUIT	1	0x00000000 00000010
CONDUIT	1	0x00000000 00000020
CONDUIT	1	0x00000000 00000040
CONDUIT	1	0x00000000 00000080
EMPTY	0	0x00000000 00000000

EMPTY	0	0x00000000 00000000
EMPTY	0	0x00000000 00000000
EMPTY	0	0x00000000 00000000

The EMPTY value must be one that is is not valid for conversion to either PortType or ConduitType; 0xffffffff will do. Replace "PGP" and "CONDUIT" with the numerical values of the corresponding enumerators.

Gen I storage of plugin software modules

Only a few different types of protocol plugins are found on Gen I systems so each type is assigned to a fixed Configuration container:

Plugin type	**Container name**
ETHERNET	1
PGP	2
Future expansion	3-9

Later a container may be used for CONFIG_FLASH, if we ever manage to make a usable wrapper for the FCI package that makes it look like another plugin.

Use case: Gen I booting

1. Boot code:
    a. Loads and starts the system core.
2. System core
    a. Initializes the CPU.
    b. Initializes RTEMS.
    c. Initializes any extra C++ support.
    d. Sets up the MMU's TLB and enables the MMU.
    e. Creates the default instance of the dynamic linker.
    f. Reads ConfigSpace.
        i. The first read triggers the reading of Configuration container zero.
        ii. Builds the Port and PortFactory lists, reading, linking and running plugin software modules as required.
    g. Performs other initialization, e.g., ethernet, BSD stack.
    h. Loads and links the application code using the default dynamic linker.
    i. Calls the application entry point.

Source code organization

All the classes, enums and other declarations will appear within the top-level namespace `ppi` and in namespaces nested within it.