# Scope

This document describes the interface between the BedComm Core and BedComm drivers. A framework called the Driver Foundation has been abstracted from the drivers to provide convenient and consistent methods and data structures for use when handing off serial data to the BedComm Core from medical devices.

# Product Description

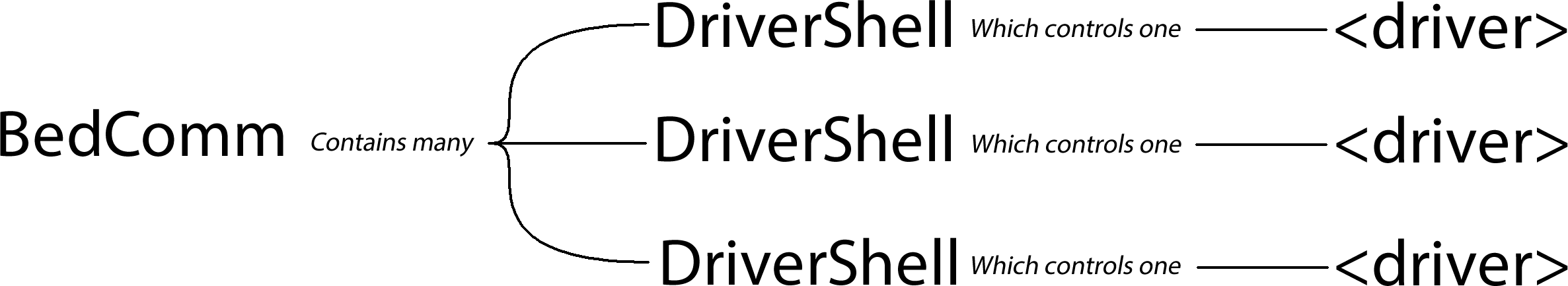
BedComm is a product containing both hardware and software. It is a tool used for aggregating medical data from other devices and/or systems; predominantly auxilliary medical devices and HL7 feeds from external servers. The primary functionality exists in what will be referred to as the “Core.” The Core consists of a dynamically linked library (DLL) and a Windows service.

# Theory of Operation

The Core is designed in such a way that it can support “drivers” and “plugins” which can be developed separately without needing to change and re-build Core software. The Core DLL contains classes and methods that can be used and extended by drivers and plugins. Drivers and plugins also exist as DLL’s that the Core Windows service can load and use.

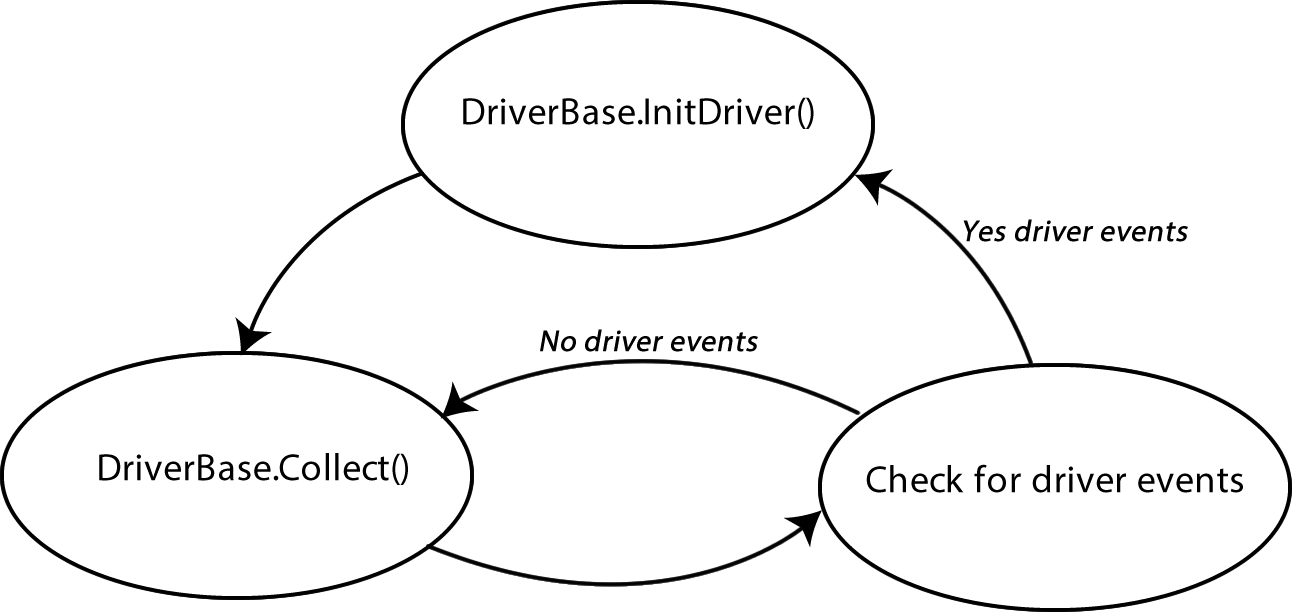
Every driver, in the context of the Core, runs in a DriverShell, as shown in Figure 1.

Figure : Core - Driver Management



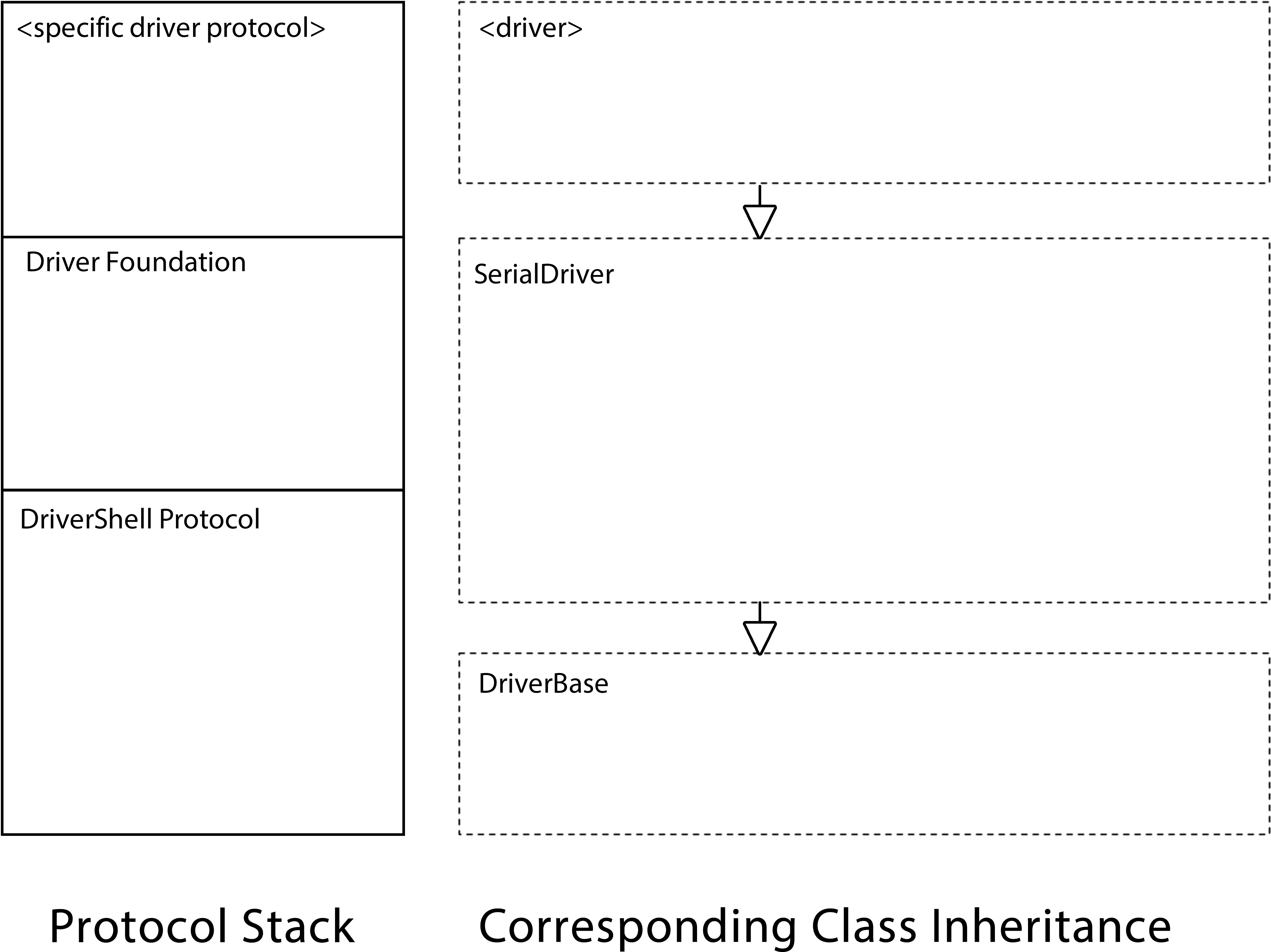
DriverShell, which inherits from “Activity,” is implemented as a state machine. The states that pertain to driver design are shown in Figure 2:

Figure : DriverShell Driver Management States



This is the basis for what will be referred to as the “DriverShell Protocol.” The DriverShell Protocol is the base definition for how a driver is supported by the BedComm Core. Since its inception, a more robust protocol based on DRY (Don’t Repeat Yourself) principles has been developed to take advantage of patterns and redundancies found across all drivers. This protocol will be referred to as the “Driver Foundation.” It can be seen in the second tier of the protocol stack shown in Figure 3:

Figure : DriverShell - Driver Protocol Stack



# General Methods

**Each driver will define the following methods so that the BedComm Core can know basic information about the driver.**

Table : General Methods

|  |  |  |
| --- | --- | --- |
| **Method** | **Description** | **Example** |
| GetCategory() | Returns the category of medical device that the driver is built for belongs to. | return "Monitor"; |
| GetTitle() | Returns the name of the device to be used by the BedComm Core. | return "Somanetics"; |
| DriverVersion() | Returns the current version of the driver. Each driver will be released as version 1.0 Major driver changes will bump the version up to the NEXT\_INTEGER.0 Minor driver changes will bump up the tenths digit | return "1.0"; |
| GetComType() | This is used by the BedComm Core to distinguish a serial driver from other driver types. This will be the same for ALL serial drivers. | return typeof(SerialCom); |

# Serial Data Receive Profiles

**Each driver will support one of the following serial data receive profiles:**

* TimeoutRead

This profile is best for reading a packet of incoming data where the end of line character(s) is/are unknown, and the number of bytes are unknown

* CountRead

This profile is best used to receive serial data when the exact number of bytes to be read is known

* EOMRead

This profile is best used to receive serial data when the end of message byte(s) is/are known

* Other

This profile may be used if none of the above profiles are appropriate for the serial communication protocol. If this is used, then receiveDataFromDevice() must be overridden in the driver.

Table : Serial Receive Methods and Fields

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Method/ Field** | **Description** | **TimeoutRead** | **CountRead** | **EOMRead** | **Other** |
| blockingTimeoutRead() | This method reads serial data until there is a timeout. | Yes | No | No | No |
| blockingCountRead() | This method reads serial data until a specific number of bytes has been read. | No | Yes | No | No |
| blockingEOMRead() | This method reads serial data until a specific byte or byte sequence has been read. | No | No | Yes | No |
| BAUDRATE | Speed of the communication (symbols per second) | Yes | Yes | Yes | Yes |
| DATABITS | Number of bits per serial packet | Yes | Yes | Yes | Yes |
| PARITY | Error checking bit | Yes | Yes | Yes | Yes |
| STOPBITS | Number of bits at end of serial packet | Yes | Yes | Yes | Yes |
| HANDSHAKE | Flow control protocol | Yes | Yes | Yes | Yes |
| initialTimeout | Maximum time to wait between device protocol packets | Yes | Yes | Yes | No |
| latencyTmeout | Maximum time to wait between bytes in a device protocol packet | Yes | Yes | Yes | No |
| roughSize | Approximate length of device protocol packet A log(n) algorithm finds the true packet length based on this starting point This value can not be zero | Yes | No | Yes | No |
| exactSize | Exact length of the device protocol packet | No | Yes | No | No |
| EOM | Byte(s) indicating the end of a device protocol packet | No | No | Yes | No |

# Driver Profiles

**Each driver will support one of the following serial data receive profiles:**

* CollectionExtraction

A CollectionExtraction driver will extract data from a medical device when the DriverShell calls the Collect() method. This driver type is optimal when long sleep periods are acceptable, no alarms will be present, and no complex protocol needs to be perpetuated.

* BackgroundThreadExtraction

A BackgroundThreadExtraction driver will extract data from a medical device continuously in a background thread. This driver type is optimal when long sleep periods are not acceptable, alarms will be present, or a complex protocol needs to be perpetuated.

* Other

This profile may be selected only if the driver foundation is to be bypassed.

*These profiles are enumerated in ExtractionType. Each driver must set the field extractionType to one of these enumerations.*

A block diagram depicting the two functional flows (CollectionExtraction and BackgroundThreadExtraction) that make up the Driver Foundation is shown in Figure 4. In this diagram, the title of each block is in the top left corner. Each block is either a function/method (ending in opening and closing parenthases), C# statement, or psuedocode. A block inside a block indicates an embedded method call, where a solid-lined arrow indicates a subsequent method call. The dotted-lined arrow indicates that a new thread will be created at that point in the sequence.

Figure : Driver Foundation

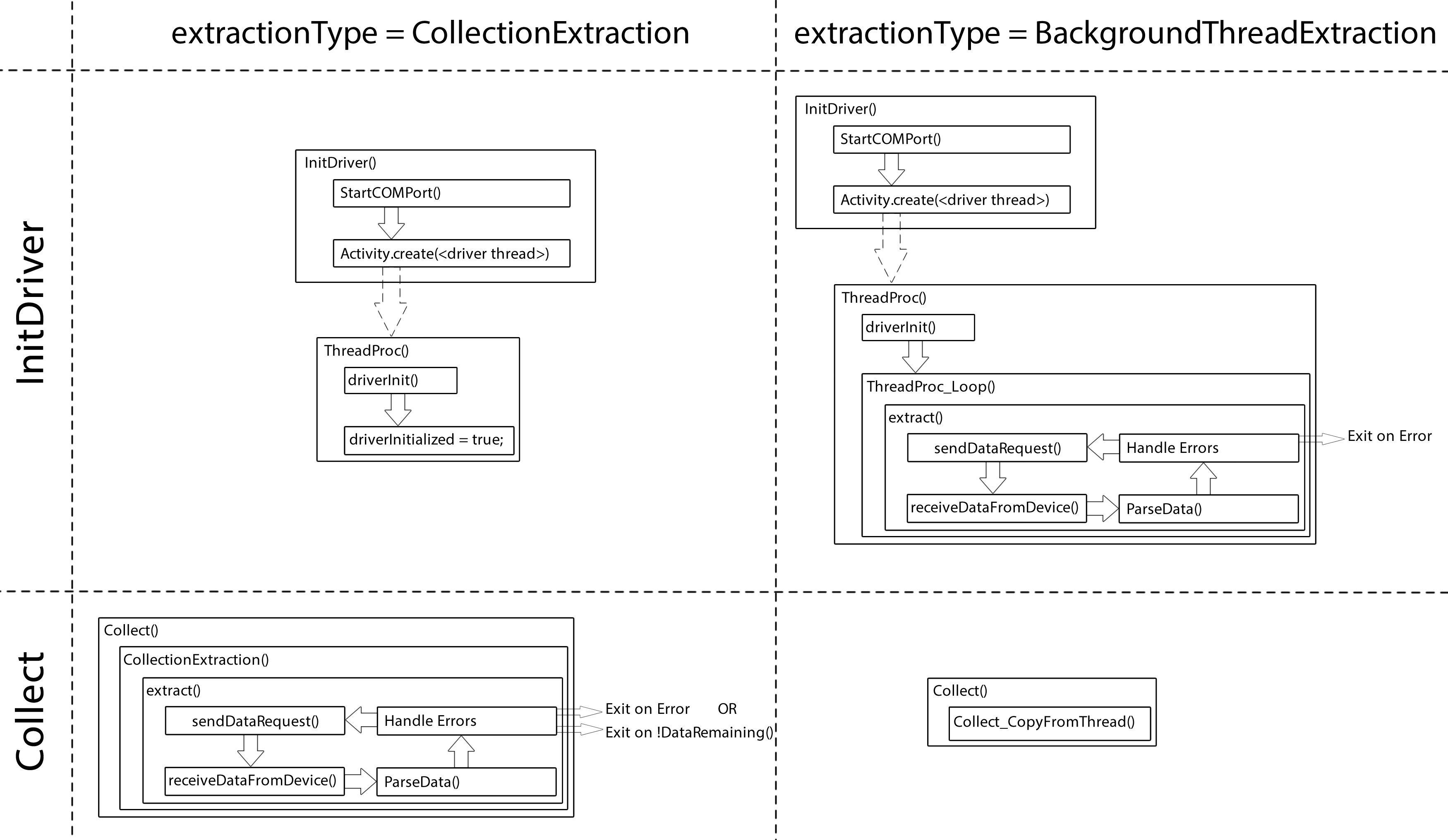


Table : Driver Foundation Methods

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Method** | **Description** | **inputs** | **outputs** | **Class containing primary logic** | **Protocol Layer** | **Collection Extraction** | **Background Thread Extraction** | **<driver> can override** | **<driver> must override** |
| InitDriver() | Called by the DriverShell, this method instantiates a driver, and creates the background driver thread if the COM port is successfully opened. | None | None | SerialDriver | DriverShell Protocol | Yes | Yes | No | No |
| StartCOMPort() | Either connects or re-connects to the COM port with the configuration specified by the driver. | None | true if the COM port connection was successful; false otherwise | SerialDriver | DriverShell Protocol | Yes | Yes | No | No |
| create(<driver thread>) | Instantiates driver, and creates the background driver thread | Out of scope | Out of scope | Activity | DriverShell Protocol | Yes | Yes | No | No |
| ThreadProc() | ThreadProc() is where the driver background thread starts. BackgroundThreadExtraction drivers enter into the extraction loop from this, while CollectionExtraction drivers set the driverInitialized flag and exit the thread peacefully. | None | None | SerialDriver | DriverShell Protocol | Yes | Yes | Bypass Driver Foundation | No |
| driverInit() | This method should be used by the driver to perform any inintialization procedures needed at driver startup. | None | None | SerialDriver | Driver Foundation | Yes | Yes | Yes | No |
| ThreadProc\_Loop() | Provides a wrapper for the extraction loop for BackgroundThreadExtraction driver types. | None | None | SerialDriver | Driver Foundation | No | Yes | No | No |
| extract() | This method provides a common framework for all drivers to extract data from a serial device. extract() is comprised of the following generic sequence of events:  1. sendDataRequest()  2. receiveDataFromDevice()  3. parseData(raw)  4. switch (driverEvent.code) If a driverEvent gets set before the step 4 switch statement, the code will fall through to the switch statement, where the driverEvent is handled. extract() should be looped forever while it returns true for BackgroundThreadExtraction drivers. extract() should be looped while it returns true and while dataRemaining() for CollectionExtraction drivers. | None | false if final driver event;  true if there was no error, or if max invalid data count has not been exceeded | SerialDriver | Driver Foundation | Yes | Yes | No | No |
| sendDataRequest() | This method should be used by the driver to transmit any commands or queries over the serial port | None | None | SerialDriver | Driver Foundation | Yes | Yes | Yes | No |
| receiveDataFromDevice() | This method is used to receive data from a serial device, and stores it into the field "raw." The receive time is recorded in the field "dataCollectTime" to be used later for timestamping data. | none | Raw bytes from serial port | SerialDriver | Driver Foundation | Yes | Yes | Yes | No |
| parseData() | This method takes the raw serial bytes from receiveDataFromDevice(), and sorts it into the DataMatrix data structure | Raw bytes from serial port | None | SerialDriver | Driver Foundation | Yes | Yes | Yes | Yes |
| Collect() | This method is called on every driver by the DriverShell periodically to obtain a result object containing data from the medical device. | Result | Device (never used) | SerialDriver | DriverShell Protocol | Yes | Yes | Bypass Driver Foundation | No |
| Collect\_Extract() | This method is called by CollectionExtraction drivers to execute an extraction cycle, and returns the result from the DataMatrix. | Result | Device (never used) | SerialDriver | Driver Foundation | Yes | No | No | No |
| Collect\_CopyFromThread() | This method is called by BackgroundThreadExtraction drivers to return data from inside the DataMatrix into a result. | Result | Device (never used) | SerialDriver | Driver Foundation | No | Yes | No | No |
| dataRemaining() | This method determines if another extraction cycle is needed because there is more data remaining to be collected. | None | True if more data is to be received from the device; false if all pertinent data has been collected by the driver | <driver> | Driver Foundation | Yes | No | Yes | No |

Table : Driver Foundation Fields

| **Field** | **Description** | **Type** | **Class containing primary logic** | **Protocol Layer** | **Collection Extraction** | **Background Thread Extraction** |
| --- | --- | --- | --- | --- | --- | --- |
| driverInitialized | This field indicates whether or not the driver is ready to be collected from. If the collect method is called before this is set to true, the driver will return a null resultl. | boolean | SerialDriver | Driver Foundation | Yes | Yes |

# Parameters and Data Structures

**Each driver will support a set of parameters that the medical device will send over the serial port.**

Each parameter will be one of the following Parameter Types:

* Data

This data type is typically measured values. These parameters are sometimes refferred to as vitals.

* Setting

This data type refers to settings that are set on the medical device. These are important to distinguish from the measured data parameters, as they do not indicate measurements of patient vitals.

* Alarm

This data type is an alarm sent from the medical device. This data type is sent from the driver asynchronously to the BedComm Core using the SendAlarm() method; ie: Collect() does not need to be called.

*These types are members of the class ParmType. Each parameter will be set to one of these types.*

Table : Parameter Methods

| **Method** | **Description** | **Inputs** | **Outputs** | **How to use** |
| --- | --- | --- | --- | --- |
| GetParmList() | All device parameter names, units of measure, types, and any mappings will be defined inside this method. | None | List of ParmInfo objects; essentially the metadata for each parameter | Every driver must have a custom override of this function. See InfoMatrix for more details of how to implement |
| SendAlarm() | This method is to be called by the driver when an alarm is to be sent. The alarm is pushed onto a queue used by the BedComm Core, which receives the alarm asynchronously | Alarm Result | true if the same alarm has been sent recently false if the alarm was only just sent by the function call | If the driver has an alarm, create a Result object, add the alarm(s) to it, and pass the Result object to a SendAlarm() function call |

Any time data is parsed from a serial device, the received data will be stored into a DataMatrix structure. Each driver will have a static structure called InfoMatrix to store metadata about paramters. MatrixKeys is used for addressing rows in each of these matrices. Finally, a structure called collected will indicate whether or not data has been collected for a given row in the matrices. These structures are illustrated in Figures Figure 5 and Figure 6.

Figure : Static Matrix Structures

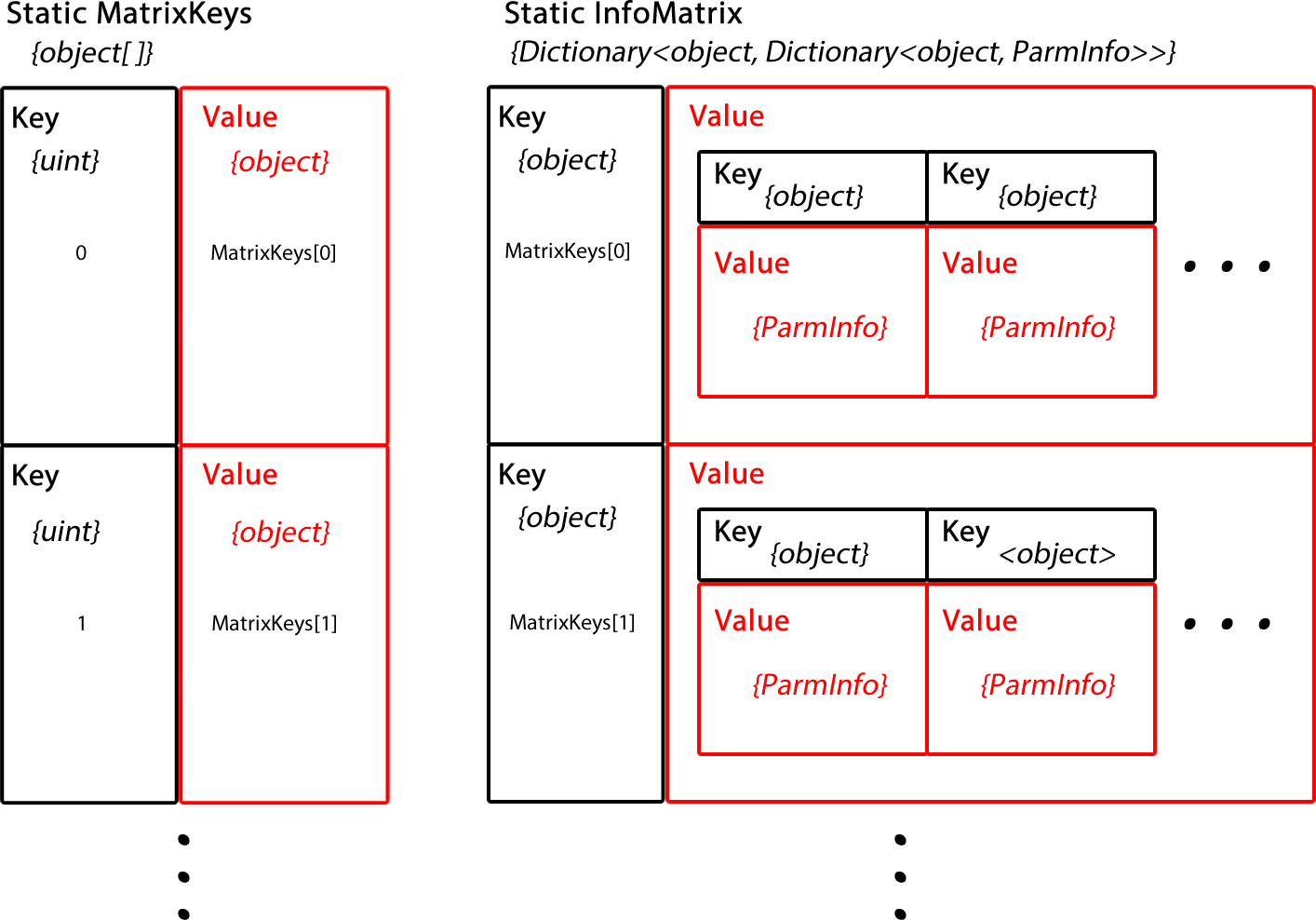


Figure : Instance Matrix Structures

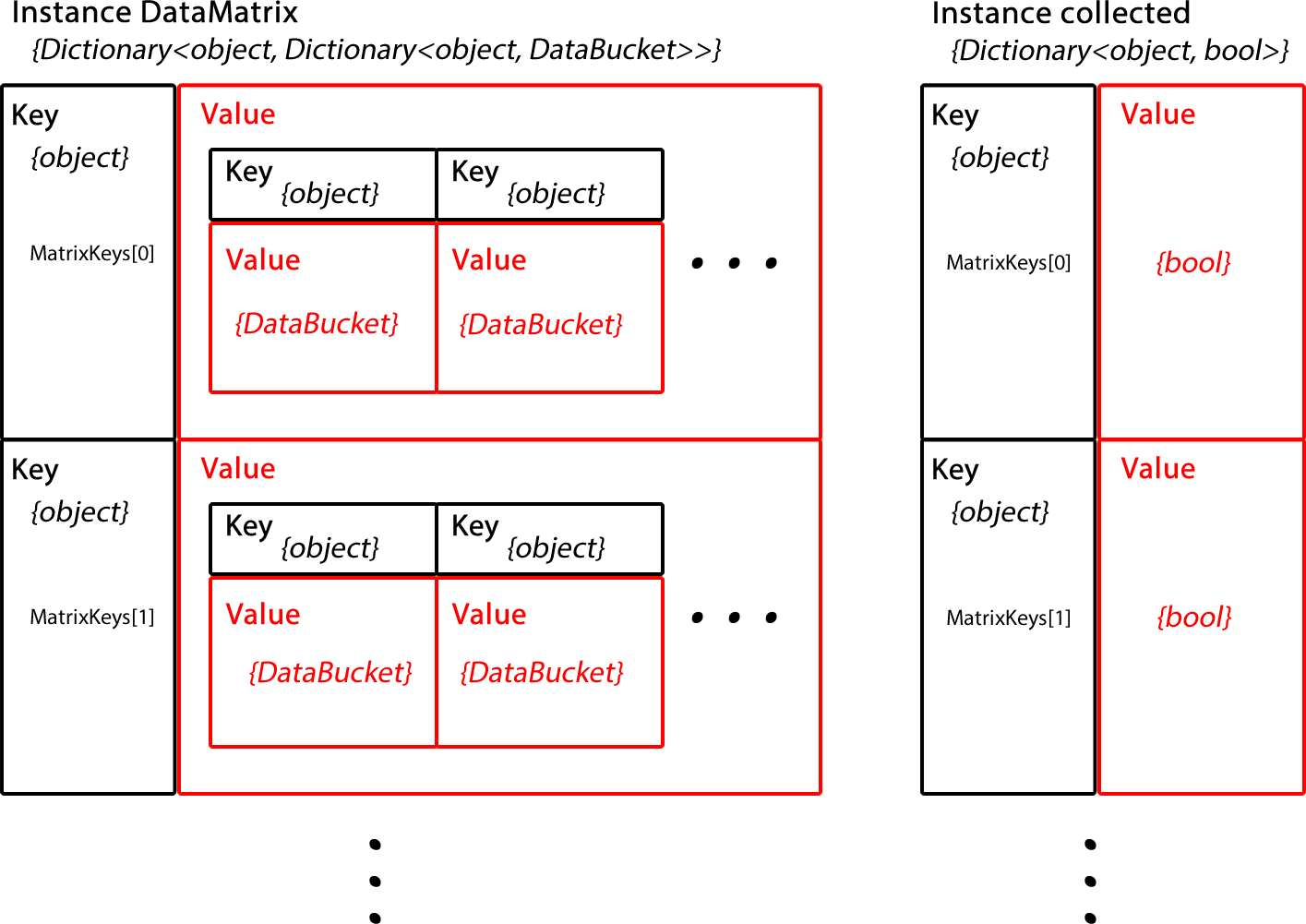


Table : Data Structures

|  |  |  |  |
| --- | --- | --- | --- |
| **Data Structure** | **Description** | **Type** | **How to use** |
| ParmInfo | ParmInfo is a class that contains only the metdata (info) about a parameter. This includes the following fields:  • ParmInfoID - <device name>.<parameter name>  • stdID - <parameter name>  • stdUnits - <units of measure>  • parmType - <parameter type> | Database Class | Each driver typically defines a method called AddParm(), which creates a ParmInfo using the most comprehensive ParmInfo constructor and adds it to the InfoMatrix. |
| ParmValue | ParmValue is a class that contains metadata about a parameter, the value of the parameter, and its timestamp. This includes the following fields:  • id - <parameter name>  • uom - <units of measure>  • val - <value>  • type - <parameter type>  • ptime - <timestamp> | Class | This class is not used directly by drivers. |
| Result | Result is a database class that contains a list of ParmValues from the driver. It where the driver hands off the baton to the BedComm Core. | Database Class | The only place that a driver needs to deal with this class is when sending an alarm; If the driver has an alarm, create a Result object, add the alarm(s) to it, and pass the Result object to a SendAlarm() function call |
| InfoMatrix | The InfoMatrix is a two dimensional dictionary that stores the metadata of all the parameters to be received from the medical device. The InfoMatrix is static member (it is a shared field between all instances of a driver class) This will store the parameter names, units, datatypes, as well as any key mappings. | Instance of  Dictionary<object, Dictionary<object, ParmInfo>> | Store metadata into this inside GetParmList() |
| DataBucket | DataBucket is a class to store all data about a single driver parameter, including all metadata, the most recent value collected from the device, and its timestamp. | Class | These variable types can be updated using the update() method |
| DataMatrix | The DataMatrix is a two dimensional dictionary that stores the latest parameters received from the medical device. The DataMatrix is an instance member that should be identical to the static InfoMatrix, except instead of consisting of ParmInfo's, it consists of databuckets (which also contain the corresponding ParmInfo) | Instance of  Dictionary<object, Dictionary<object, DataBucket>> | Update parameters as they are parsed out the the received serial data. Use the update() method. |
| MatrixKeys | The InfoMatrix and DataMatrix are both two dimensional dictionary structures. MatrixKeys is a collection of the keys to the addresses of the inner dictionaries, (they allow the selection of a 1 dimensional dictionary row out of a 2 dimensional dictionary grid). With the combination of a "Matrix Key" and a "Parameter key", a specific ParmInfo may be selected out of the InfoMatrix, or a specific DataBucket may be selected out of the DataMatrix. | Instance of object[] | There must be one "Matrix Key" for each set of parameters that are received differently in the serial protocol. The keys may be increasing integers, or a protocol specific key mapping. |
| collected | This field indicates whether or not a particular row of the DataMatrix has been collected. | Instance of Dictionary<object, bool> | Indicate that a row in the DataMatrix has been updated by setting its respective location in the array to true. In dataRemaining(), return false only when all elements of this array are set to true. |

# Options

**Each driver will have the following configurable fields to further customize how the driver functions:**

Table : Options

| **Field** | **Type** | **Description** |
| --- | --- | --- |
| sendBlankResults | boolean | Whether or not to send blank parameter values in a result |
| clearUponCollection | boolean | Whether or not to clear the DataMatrix upon the Collection method call |
| dbug | boolean | Whether or not computationally intensive string concatenations occur inside debug statements will depend on this flag |
| WriteReceivedDataToFile | boolean | Whether or not to write every raw packet to a separate hex file in C:\BCSimulatorData\<driver name> |
| MaxInvalidDataCount | integer | MaxInvalidDataCount indicates the maximum tolerable invalid data packets that can be received |
| backgroundThreadSleep | integer | backgroundThreadSleep indicates the number of milliseconds between each extraction cycle |

# Error Handling

**Each driver will notify the BedComm Core of a major driver event asynchronously.**

Every driver event will have one of the following event codes:

* NO\_ERROR

This is a non-fatal transient driver event code that only exists at driver instantiation, and before the completion of the first extraction sequence. If the first extraction sequence was successfull, the driver event status changes to VALID\_DATA. If the first extraction sequence was unsuccessful, the driver event is set to it's respective event code.

* VALID\_DATA

After the first successful extraction sequence, the driver event is set to this non-fatal code, and the VALID\_DATA event is sent to the BedComm Core using the SendError() method.

* INVALID\_DATA

This semi-fatal driver code is set when the received serial data can not be parsed, the data did not match an expected signature, or the checksum/CRC byte(s) was/were not correct. This event may happen multiple times before exiting the extraction loop and sending a driverEvent; the maximum number of times this can happen is set by MaxInvalidDataCount.

* NO\_DATA

This fatal driver code occurs when a timeout occurs before any data is received.

* PORT\_ERROR

This fatal driver code occurs when a the COM port passed to the driver can not be opened.

* OTHER\_ERROR

This fatal driver code occurs when one of the above error codes do not apply, or when an unknown error occurs.

*These event codes are enumerated in DriverEventCode Each driverEvent will be set to one of these event types*

Table : Driver Event Items

|  |  |  |  |
| --- | --- | --- | --- |
| **Member/Code** | **Description** | **Function** | **How to use** |
| driverEvent | This field contains the current driver event status. | Once the driverEvent is set, the Driver Foundation will terminate any background threads, and send the driver event to the BedComm Core using a SendError() method call in SerialDriver. | All that the driver must do is set the driver event to a fatal event code, and the driver will terminate properly. |
| SendError() | This method is called at specific locations defined by the Driver Foundation in order to send a single alarm when driverEvent gets set anywhere in the driver. This method does not need to be used when writing a driver; it will only be called in the SerialDriver class. | This method sends the current driverEvent to a queue being read by the BedComm Core. The method will return false if the driver event has already been sent; true if it has just been successfully sent by the function call. | This method should not be called by the driver. It is called in SerialDriver in order to push the driver event onto a queue that the BedComm Core is constantly reading from. |
| switch (driverEvent.code) | This is a section of code that handles driver events inside the extract() method. | This section of code is referred to in diagrams as "Handle Errors." It signals through the extract() method's return object whether or not to continue the extraction cycle. | This code exists inside the SerialDriver class, and thus, should not be modified when writing a driver. |