```
REP: XXX
Title: ROS-Industrial Simple Message Generic Robot IO
Author: G.A. vd. Hoorn <g.a.vanderhoorn@tudelft.nl>
Status: Draft
Type: Standards Track
Content-Type: text/x-rst
Created: 01-May-2014
Post-History: 01-May-2014, 27-May-2014, 13-Aug-2014
```

Outline

Abstract

This REP describes extensions to the ROS-Industrial Simple Message protocol for controlling the generic IO functionality present on industrial robot controllers. It specifies additional message structures used for directly reading and writing IO elements, as well as for setting up state streaming. Finally, it discusses alternatives to the proposed structures and provides arguments for why they were rejected.

Motivation

In order to meaningfully integrate an industrial controller into a ROS application, two main functional areas of those controllers need to be accessible: motion control and input-output. Since its inception, ROS-Industrial has provided an interface to the motion control functionality through the Simple Message [1] protocol. Although robot specific interfaces were created [2] [3], ROS-Industrial drivers have been lacking generic, reusable access to the IO facilities provided by controllers [4].

A key principle in the design of such an interface would have to be the reusability of the client components. Simple Message made this possible for motion by standardising the protocol used for communication between the client and the servers. Driver authors need only implement support for this protocol to be able to reuse the ROS nodes made available in the *industrial_robot_client* package [5].

This REP seeks to extend the defined set of Simple Message structures with new messages and interaction patterns in order to create such a reusable interface, similar to what exists for controlling motion on supported industrial controllers.

Definitions

Controller
: The device which provides access to the Robot IO interfaces. Typically the same as the one that controls manipulator motion and / or network connectivity.

Server
: The part of the ROS-Industrial driver running on the controller that provides clients with access to the IO facilities.

Client
: ROS-Industrial node that connects to the server and provides a ROS node graph with access to the IO facilities of the controller through the implemented ROS API.

Robot IO
: All (electrical) interfaces into and out of the controller that an operator can use to interact with and sense the world outside that controller. Examples are digital and analogue IO. Note that on some controllers the name *Robot IO* is given to a specific subset of the available IO facilities. This REP does not follow that convention, but instead uses the name to refer to all IO Elements on a server.

IO Element
: Smallest, discrete unit of IO that is addressable (whether with Indexed or Named IO) for a given IO type. For digital IO, this would correspond to a single pin. For analogue IO, this would correspond to

a single analogue input or output terminal on the controller.

Indexed IO
: Type of Robot IO in which IO Elements are accessed by writing to or reading from an element in a predefined (global) array or other form of indexable structure. Typically, the base type of this array corresponds to the type of IO: Floating point for analogue IO, Boolean or Integer for digital. Numerical indices are used to address specific elements.

Named IO
: Type of Robot IO in which IO elements are accessed by writing to or reading from predefined (global) named variables. Only the predefined variables can be referenced in program statements, with other IOs inaccessible. Typically, the runtime system provides the operator with functionality to create, name and map variables onto physical IO locations on the controller.

Assumptions

1. In all message structure definitions, *reading* shall be understood to transfer data from the server to the client. *Writing* is the complementary action, transferring data from the client to the server.
2. This REP uses zero-based numbering on all indexable fields. Array indices start at 0. Drivers for controllers that use a different convention will have to take this into account.
3. Default values for message fields are zero (0) for integer, Boolean and floating point fields, and the zero-length string for text fields. Unused bits (in bitmasks) should be set to zero (0).
4. All string fields shall use Pascal strings (string length encoded as a single byte, followed by character data).
5. No specific byte-order is prescribed in this REP, although driver authors are encouraged to use little-endian order for their network communications. The ROS generic IO node shall be able to perform byte-swapping if needed.
6. Digital IO Elements are assumed to be representable by single-bit integer values (ie: Booleans). For Analogue IO Elements floating point values are assumed (ie: Floats). Complex IO types will require special care during (de)serialisation by both the client and the server.
7. None of the message structures in this REP include any fields or other substructures for use in message payload validation. A reliable transport layer is assumed for all communication between client and server. In cases where this cannot be guaranteed, driver authors are expected to include sufficient error detection and recovery mechanisms at the application layer level, to be able to deal with any communication errors.

General Design

This section describes the general design of the Generic IO protocol extension.

Requirements

Design of the protocol extension described in this REP was guided by the following high-level requirements:

1. Should try to be efficient in its use of controller resources.
2. Should support reading and writing of the types of IO most commonly found on industrial controllers: digital, analogue, grouped.
3. Should try to minimise required user configuration and maintenance on the server side.
4. Should make switching between manipulators from different brands relatively easy (ie: no recompilation or rewriting of code).
5. Should support extension and specialisation by users.
6. Should not require advanced, dynamic runtime behaviour for its basic functionality (ie: be implementable with the constraint native languages supported by industrial controllers).

Requirements 1 and 6 imply the use of simple (statically defined) structures with simple data types. Requirements 3 and 4 suggest the inclusion of auto-negotiation of features between client and server. Requirement 5 can be fulfilled by allowing for sufficient unused message and (sub)type IDs.

Design Overview

The main purpose of the Simple Message IO protocol extension is to allow access to the IO facilities of an industrial robot controller. Similar to the motion interface, this will be facilitated by running a server component on the controller, and a client component on the ROS side. The IO server component – implemented in the native language of the controller – could either be made an extension to an already running ROS-Industrial server component, or be made to run as a separate task. In all cases, the server is responsible for listening for incoming IO messages, parsing their payload and performing the requested IO operations. Results are sent back to the client, which will relay them to the ROS node graph.

The core of this extension is a pair of synchronous read and write operations, combined with a server side initiated IO state publication mechanism that is configurable by the client.

This extension does not implement a bi-directional IO memory synchronisation protocol: the ROS node, while referred to as the *client* in this REP, initiates all IO operations and controls the actions of the server.

The state of the server's IO Elements is transferred to the client. Changes to that state are only performed by the server on the client's request.

Profiles

The IO operations described in this REP have been grouped into *profiles*: sets of related functionality that servers may implement support for. Not all profiles are required: a driver can claim generic IO capabilities when at least the minimum set of operations – synchronous read and write, the *Basic* profile – has been implemented. This approach was chosen to ensure that the extension could be supported on as wide a selection of industrial controllers as possible, even if some profiles may require interaction with IO interfaces in ways not all controllers allow (those profiles will then not be supported by the controller).

Two additional optional profiles are defined in this REP: *Reset* and *Streaming*. The first makes it possible to reset IO Elements to their default state (which is controller defined). The second allows the client to request continuous updates on the state of (a subset of) the server's IO Elements, without any client side polling. The server uses asynchronous publications, similar to the joint state reporting feature of the motion interface in Simple Message.

Feature Discovery

To avoid the client reporting functionality to the ROS node graph that servers do not support, and to allow clients to perform automatic discovery and configuration of supported IO interfaces, the Basic profile requires servers to include support for a feature discovery mechanism. Upon request of the client, the server returns information on the types of IO interfaces it supports, the number of configured Elements and their directionality (in, out). Apart from auto-initialisation of the client, this also allows for comparison of actual against expected manipulator configuration.

As many controllers do not require configured IO indices to be contiguous, the server reports on groups of Elements, collected into *ranges*, instead of individual Elements. Each range has an associated feature descriptor, which encodes whether the server supports a certain feature or not for that range (ie: analogue Elements may be resettable, digital may not be). In addition, a global descriptor is used to report on features that are not range or type specific.

Addressing

IO Element addressing is based on `(type, index)` tuples, in which both elements are numerical indices. Servers use a look-up table to implement the mapping between `type` identifiers and the IO interfaces they represent (Generic IO Type Identifiers). This table is expected to be small and to be static (as supported IO interfaces are only expected to change when the (hardware) configuration of the controller changes). Element indices – within IO types – are unique, with support for gaps provided by ranging.

Support for named Elements is not part of the protocol extension described in this REP. See Default to Named IO in the Alternatives section for the rationale behind this design choice. Semantic naming of IO Elements will be addressed at the ROS API level.

Named IO Controllers

The design for servers running on controllers that provide only name based access to their IO Elements is slightly different.

As these controllers do not allow access to unconfigured (ie: anonymous and unlinked) IO Elements, a set of named Elements will be predefined by the driver author, which will then be linked to controller IOs at configuration time by the user (note 1). The names of the predefined Elements will reflect their types (fi: `doutN`, `ainN`, with `N` a positive integer), insofar as this is not already enforced by the software running on the controller.

A set of look-up tables embedded in the server can now be used to map between the types and indices specified in the message structures and the corresponding named IOs on the controller (note 2). For example, a read request of Element `5` of type `3` should be mapped to a read of the named IO `ain5` (see Generic IO Type Identifiers).

Requests by the client for the configuration of the server would return the hard-coded set of predefined types and indices, based on the predefined named variables (note 3). This would let the Named IO controller essentially emulate an Indexed IO controller, albeit with a (likely) smaller IO space (note 4). It would be the responsibility of driver users to make sure that all predefined IOs are actually correctly setup (ie: linked to controller IOs). If possible, driver authors should strive to make servers capable of dynamically detecting the number of correctly setup (ie: linked) predefined IOs, and report only those to the client. Alternatively, driver authors may provide users with a controller based configuration interface that allows users to manually provide this information.

The abstraction provided by this design shields the generic IO client from the inherent differences between Named and Indexed IO servers. The remainder of this document therefore will only define interfaces to Indexed IO servers.

Notes

1. Obviously, the described setup does not require the user to configure a *contiguous* range of system IO Elements to be used for predefined application IOs, although such a configuration would best mimic

Indexed IO controllers.
2. Depending on controller support, instead of look-up tables the naming scheme could be exploited to dynamically determine the mapping between indices and named IO. This would increase flexibility of the server, as increasing the number of available named IOs would not require updating of the look-up tables.
3. See the `IO_INFO` message in the Basic Profile section.
4. While in theory there is no limit on the number of named IOs that could be predefined, practically, the set should be kept rather small, as large sets could significantly increase the time spend configuring the controller and application.

Message Definitions

The following sections describe the message structures that make up the Simple Message generic IO protocol extension. The messages have been grouped into *profiles*, of which only the *Basic* profile is mandatory. The *Reset* and *Streaming* profiles are optional, however driver authors are encouraged to add support for at least the Streaming profile to their drivers.

Overview

The following message structures are defined in this section:

```
Basic Profile
  IO_INFO
  IO_READ
  IO_WRITE

Optional Profiles
  Reset
    IO_RESET

Streaming
  IO_STREAM_SUB
  IO_STREAM_UNSUB
  IO_STREAM_PUB
  IO_STREAM_CFGGET
  IO_STREAM_CFGSET
```

ROS-Industrial Packet Header

The following structure describes the *ROS-Industrial packet header* (copied from [6]):

```
prefix
  length
header
  msg_type
  comms_type
  reply_code
```

Whenever subsequent message definitions refer to this header, its contents (named fields and their types) should be considered to be part of the definition, and the value of the `length` field should reflect the total number of bytes in the defined message.

Basic Profile

All generic IO servers shall support the messages defined in this section. The *Basic Profile* provides a minimal interface to the IO facilities of a controller, which lets clients perform synchronous read and write operations (akin to an RPC invocation).

Servers shall use the `IO_INFO` message to indicate support for any additional features they might support.

IO_INFO

On reception of this message, servers shall return information on the types of IO supported, and their configured ranges. Consecutive Element indices of the same type shall be grouped into ranges, to limit the size of the reply and to efficiently handle large numbers of IO Elements.

Range descriptors also indicate support for optional features, such as change notifications. Feature descriptors have per-IO-type granularity (ie: a controller may support change notifications for digital IO, but not for analogue or grouped IO).

This is a read-only message. Controllers that cannot retrieve information about their IO facilities at run time may opt to hard-code a minimum support profile into their driver. Alternatively, driver authors may include a (run time) configuration interface on the controller which would enable a user to update the information to be returned, based on the physical configuration of the controller.

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id        : uint32
```

Reply:

```
ROS-I Header
message_id        : uint32
ctrlr_feat_mask   : uint32
num_items         : uint32
items[]
{
  type            : uint16
  start           : uint16
  len             : uint16
  feat_mask       : uint32
}
```

Defined bit positions in `ctrlr_feat_mask` are:

```
Pos  Description

  0  Controller Local Timestamps Support
```

All other positions are reserved for future use.

Defined bit positions in `feat_mask` are:

```
Pos  Description

  0  IO Reset Support
  1  Streaming Support
```

All other positions are reserved for future use.

Errors

- None

Notes

1. The `message_id` field shall be copied unchanged by servers from requests to replies. Clients can use this field to differentiate between subsequent messages. Its use is not mandatory. If left empty by a client, servers shall also leave the field empty in the reply.
2. The `num_items` field shall encode the number of repeated entries in the `items` array. In most cases, this number will be different from the byte length of the array, which can be calculated as `num_items * (num fields * sizeof(each field))`.
3. Refer to section Generic IO Type Identifiers for a list of all defined values for the `type` field.
4. The `start` and `len` fields fully define an IO range: the `start` field denotes the index of the first IO Element in the range, the `len` field denotes the number of IO Elements in the range. A range with a length of 1 addresses a single IO Element.
5. It is an error to return overlapping ranges (ie: ranges for which the `start` index lies within another range).
6. Element indices must be globally unique within their IO type: no two ranges may contain the same `start` index, or contain indices from other ranges.
7. The `ctrlr_feat_mask` and `feat_mask` fields are bitmasks. A bit with a value of 1 indicates support for a feature. All other bit positions are reserved, and should be initialised to zero. Bit positions are counted starting from LSB.

IO_READ

Requests the controller to read the value of the specified IO Element(s).

It is allowed for one read request to access multiple different indices, each reading Elements of a different type (example: two digital IOs, one analogue).

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type     : uint16
  index    : uint16
}
```

Reply:

```
ROS-I Header
message_id : uint32
timestamp  : uint32
num_items  : uint32
items[]
{
  type     : uint16
  index    : uint16
  result   : uint16
  value    : uint32
}
```

Errors

- IO type not supported (1001)
- Invalid index: out of bounds for IO type (2001)

Notes

1. Refer to section Status and Error Codes for a list of all defined error and status codes.
2. The value of the `result` field shall reflect the outcome of the requested operation. If not equal to `SUCCESS`, the *Errors* subsection of each message definition lists the possible return values. For messages that support batching of operations, the result of each individual operation shall be reflected by the `result` field in the returned array, in addition to the `reply_code` field in the ROS-I header. Whenever both a per-type and a per-Element return value is defined by this REP, the per-Element value should be returned.
3. The value of the `value` field is undefined in case the read operation was unsuccessful (ie: `result` is not equal to `SUCCESS`).
4. The actual type (ie: *Integer* or *Float*) of the `value` field depends on the value of the `type` field (see section Generic IO Type Identifiers). In all cases, the server shall serialise the value of the requested IO Element to the 4 byte field, and the client casts to the appropriate type on reception of the message.
5. For IO Elements with a representation with a lower number of significant bits than the number reserved for the `value` field in the

reply, the server shall align the lowest significant bit (LSB) of the IO-value with the LSB of the `value` field (for example: 12-bit analogue IO values are serialised with their LSBs at bit 0 of the `value` field).
6. As controller support is expected to be minimal, this REP does not set a requirement for batched reads of multiple IO Elements to be *instantaneous* or to be executed at exactly the same point in time (although driver authors are encouraged to minimise delays). Clients that require snapshot behaviour could potentially issue read requests to *grouped inputs*.

IO_WRITE

Complementary to the `IO_READ` message. On reception of this message, servers shall set the value of the specified IO Elements to the provided value.

Writes to multiple Elements may be batched into one message with multiple elements and send in a single request. Writes to multiple different types of IO in a single request are also permitted (for example: two digital IOs, one analogue).

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type      : uint16
  index     : uint16
  value     : uint32
}
```

Reply:

```
ROS-I Header
message_id : uint32
timestamp  : uint32
num_items  : uint32
items[]
{
  type      : uint16
  index     : uint16
  result    : uint16
}
```

Errors

- IO type not supported (1001)
- Invalid index: out of bounds for IO type (2001)
- Invalid value: out of bounds for IO type (2002)

Notes

1. Refer to Note 4 of `IO_READ` for how the `value` field of the request should be handled.
2. Refer to Note 5 of `IO_READ` for how smaller types should be serialised into the `value` field of the request.
3. Note 6 of `IO_READ` is also applicable to `IO_WRITE`.

IO Reset (optional)

Servers that support this optional interface allow clients to request IO Elements be returned to their default state. Such a reset could be part of an error recovery strategy, or a normal start-up routine.

IO_RESET

Upon reception of this message, servers shall reset the state of the specified IO Elements to their default values. This default may for instance be the value Elements have after initial start-up of the controller, or a value that is configurable by the user on the controller itself.

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type      : uint16
  index     : uint16
}
```

Reply:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type      : uint16
  index     : uint16
  result    : uint16
}
```

Errors

- IO type not supported (1001)
- Feature not supported by IO type (1002)
- Feature not supported by IO index (1003)
- Invalid index: out of bounds for IO type (2001)

Notes

1. Clients may request a reset of all IO Elements by sending a request with a single (1) element, specifying a value of `0xFFFF` for both the

type and the index fields.
2. Clients may request a reset of all defined ranges for a specific IO type by sending a request with a value of 0xFFFF for the index field, and the type field set to the value corresponding to the IO type (see Generic IO Type Identifiers).

Streaming Interface (optional)

Servers that support this optional interface shall be able to publish the state of their IO Elements at a certain rate, without the need for an explicit polling cycle initiated by the client (using IO_READ).

As it is infeasible to publish the state of *all* IO Elements continuously, a subscription mechanism will be provided, allowing clients to notify the server of their interest in specific IO Elements, which will then be included in the publication.

IO_STREAM_SUB

Request the inclusion of a range of IO Elements in the state publication report periodically send out by the server.

Clients may subscribe to subsets of Elements within defined ranges, as described by the IO_INFO reply message. Multiple subscriptions may be batched into one request.

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type     : uint16
  start    : uint16
  len      : uint16
}
```

Reply:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type     : uint16
  start    : uint16
  result   : uint16
}
```

Errors

- IO type not supported (1001)

- Feature not supported by IO type (1002)
- Feature not supported by IO index (1003)
- Invalid index: out of bounds for IO type (2001)

Notes

1. A subscription to a range that starts at index 4 and has length 2 will result in IO Elements 4 and 5 to be added to the `IO_STREAM_PUB` payload.
2. Clients may subscribe to a range with a `len` equal to 1 in order to stream individual IO Elements.
3. Servers should fail requests for subscriptions to ranges which include IO Elements not contained in any of the defined IO ranges (as reported by the `IO_INFO` response message). Clients should avoid trying to subscribe to IO Elements not contained in any valid range by checking against known ranges prior to sending a request.

IO_STREAM_UNSUB

Upon reception of this message, servers shall cancel the subscriptions for the specified ranges. Any subsequent `IO_STREAM_PUB` messages shall not include the state of Elements in the unsubscribed ranges. Multiple ranges may be unsubscribed in one request.

Clients shall identify ranges by providing the index of the IO Element that forms the start of the range, and the IO type for each range.

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type      : uint16
  start     : uint16
}
```

Reply:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  type      : uint16
  start     : uint16
  result    : uint16
}
```

Errors

- IO type not supported (1001)
- Invalid index: out of bounds for IO type (2001)

- Invalid index: no known subscription (2003)

Notes

- None

IO_STREAM_PUB

This message shall be send by the server at a certain rate to publish the state of all IO Elements in which the client has expressed an interest. Based on the number of subscriptions, messages may contain the state of multiple Elements (potentially of different types).

Message type: *asynchronous publication*

Msg:

```
ROS-I Header
timestamp : uint32
num_items : uint32
items[]
{
  type     : uint16
  start    : uint16
  len      : uint16
  values[]
  {
    value : uint32
  }
}
```

Errors

- None

Notes

1. The `timestamp` field should be set to the value of a local (high resolution) clock by servers that support this (see `IO_INFO`). Servers without such support should initialise the field to 0.
2. Refer to Note 4 of `IO_READ` for how the `value` field of the request should be handled.
3. Refer to Note 5 of `IO_READ` for how smaller types should be serialised into the `value` field of the request.

IO_STREAM_CFGGET

Upon reception of this message, servers shall return the value of the specified item from the current streaming configuration.

Multiple configuration items may be accessed in one message.

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id : uint32
num_items  : uint16
items[]
{
  item      : uint16
}
```

Valid IDs for the `item` field are:

```
ID       Type     Description
     1   Integer  Global publish period (us): determines publication
                  rate used for all ranges.
```

All other IDs are reserved for future use.

Reply:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  item      : uint16
  type      : uint16
  result    : uint16
  value     : uint32
}
```

Defined IDs for the `type` field are:

```
ID        Description

      1   Boolean
      2   Integer
      3   Floating point
      4   String
```

All other IDs are reserved for future use.

Errors

- Invalid index: no such item (3001)

Notes

1. Refer to Note 4 of `IO_READ` for how the `value` field of the request should be handled.

IO_STREAM_CFGSET

Upon reception of this message, servers shall set the value of the specified items to the specified values.

Updates to multiple configuration items may be send in one message.

Message type: *synchronous service*

Request:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  item     : uint16
  type     : uint16
  value    : uint32
}
```

Reply:

```
ROS-I Header
message_id : uint32
num_items  : uint32
items[]
{
  item     : uint16
  result   : uint16
}
```

Errors

- Invalid index: no such item (3001)
- Invalid value: out of bounds for item (3002)

Notes

1. See `IO_STREAM_CFGGET` for valid values for the `item` field.
2. See `IO_STREAM_CFGGET` for valid values for the `type` field.
3. Refer to Note 4 of `IO_READ` for how the `value` field of the request should be handled.
4. Refer to Note 5 of `IO_READ` for how smaller types should be serialised into the `value` field of the request.
5. Changes to configuration items may take effect after a delay of one (1) publication cycle.

Status and Error Codes

This section defines status codes and associated error messages:

```
ID             Description

            0  Reserved

            1  Success

       2-1000  Reserved for future use

         1001  IO type not supported
         1002  Feature not supported by IO type
         1003  Feature not supported by IO index

    1004-2000  Reserved for future use

         2001  Invalid index: out of bounds for IO type
         2002  Invalid value: out of bounds for IO type
         2003  Invalid index: no known subscription
```

2004-3000	Reserved for future use
3001	Invalid index: no such item
3002	Invalid value: out of bounds for item
3003-64000	Reserved for future use
64001-65000	Manufacturer specific
65001-65535	Freely assignable

The IDs allocated to the *Manufacturer specific* range may be used by driver authors to add error messages that are too specialised to be included in the generic IO client. Note that the generic IO client will not be able to decode these IDs, and driver authors are expected to provide an extended version of the client able to decode messages with manufacturer specific IDs.

All IDs allocated to the *Freely assignable* range may be freely used by users and allows for ID assignment within a limited scope (ie: per project).

Generic IO Type Identifiers

This section defines numeric identifiers for IO types, to be used in the IO `type` fields:

ID	Description
0	Reserved
1	Digital In
2	Digital Out
3	Analogue In
4	Analogue Out
5	Grouped In
6	Grouped Out
7	Flags
8-64000	Reserved for future use
64001-65000	Manufacturer specific
65001-65535	Freely assignable

The IDs allocated to the *Manufacturer specific* range may be used by driver authors to add support for types of IO that are too specialised to be included in the generic IO client. Note that the generic IO client will not be able to decode these IDs, and driver authors are expected to provide an extended version of the client able to decode messages with manufacturer specific IDs.

All IDs allocated to the *Freely assignable* range may be freely used by users and allows for ID assignment within a limited scope (ie: per project). An example would be IO messages for custom end effector hardware that do not fall into any of the already defined categories.

ROS API

A suitable ROS API will be defined in a separate REP. This API will define topics and services that can be used to interact with drivers that have implemented the IO interfaces specified in this REP.

Alternatives

This section reasons about alternative design choices and why they were rejected.

Default to Named IO

As the most obvious alternative, Named IO would use strings to address the IO Elements on the controller. No type or index information would need to be communicated, as unique strings can unambiguously identify an IO Element on the server.

The main advantage of this would be the possibility to directly map IO Element names to topics and services, removing the need for any look-up tables on client or server. In addition, the names would provide an abstraction layer between the physical IO point (ie: actual pin on an IO terminal) and the logical entity used to address it. Such a layer would facilitate switching between controllers, as IO references can be rewired by simply changing names and connections.

There are however several significant drawbacks to the use of a name based addressing scheme – at the Simple Message layer – which lead to this alternative being rejected.

First, depending on the actual naming scheme, named references can significantly increase the payload sizes of Simple Message structures. For example: a read of an IO Element named *GripperClosed* would require `13` bytes (just the identifier, not including other fields). Using Indexed IO, this read would serialise to `2 + 2 == 4` bytes (for the `type` and `index` fields). Choosing a naming scheme as described in the Named IO controllers subsection of the General Design, would result in names such as `din4` and `aout10`. While this would reduce serialised message size, it also negates the main advantage of Named IO: support for semantic naming.

Second, not all industrial controllers support string parsing operations sufficiently or are able to deserialise strings efficiently from network traffic. String parsing is also inherently slower than dealing with integers.

Third, as Indexed IO controllers do not support named references to IO Elements, driver authors would have to implement a look-up table and provide a means for users to keep that table up-to-date (although the same is true for Named IO controllers in the current design).

Fourth, many Named IO controllers impose certain constraints with respect to naming IO Elements. Maximum length limitations and restrictions to certain character set are common. The names would have to be exported to the ROS registry by the client, which could lead to incompatibilities with established naming guidelines [8] [9]. Re-using IO Element names from one controller to another would also be problematic, unless a smallest common denominator approach is used.

Fifth, many Named IO controllers do not support run-time, string based addressing of their IOs, but only allow static, compile-time global variable references to be used (rewiring is typically used in such cases to allow users to connect names to existing IO Elements on the controller). This restriction would make it impossible to implement a server application that dynamically addresses IO Elements based on message contents without relying on a look-up table again.

Finally, the main advantages of Named IO – semantic naming, rewiring – can be supported on the ROS API level, either with currently existing mechanisms in ROS or with new functionality provided by the client(s).

Named IO with Indexed Protocol

The second alternative considered is a hybrid between Named and Indexed IO. In this alternative, names are used to address individual IO Elements, but in order to avoid the increase in message payload sizes, the client places only numerical IDs and indices in the serialised payloads (such as those defined in Generic IO Type Identifiers). The mapping between names and indices would be done via a look-up table on the client side, using `name → (type, index)` pairs. Names recognised by the client are those configured on the server.

While this approach would allow all of the benefits described in the first alternative, without the associated increase in message sizes, the use of an additional look-up table would increase maintenance effort for users, as they have to keep this table up-to-date, as well as the table in use on the server. In addition, it only deals with the increase in payload size, and does not address any of the other issues.

Single 'IO Operation' Message Structure

Instead of separate messages for synchronous read and write, a single message structure is defined, in which every element of the `items[]` array has one additional field: `sub_cmd`. The value of this field determines whether a read or a write is performed. A single message payload may combine multiple reads and writes. This alternative could be used with both Indexed and Named IO.

This alternative was rejected due to the fact that combining all IO operations into one message, and relying on sub IDs to discriminate between operations is not substantially different from using individual message types (with support for batched operations). It is therefore not expected to provide any significant benefits over the design described in General Design. Message parsing would however be slightly complicated by the differences between read and write operations and message sequences would become less explicit as intent is encoded by a (sub)field only.

Additionally, it is unclear whether the temporal distribution of reads and writes will ever allow for them to be combined into one message (without resorting to buffering on the client), leading to multiple messages being used and negating any potential decrease in network traffic.

Finally, a combination of synchronous writes and IO streaming would seem to cover the same use cases.

Backwards Compatibility

This REP extends the simple message protocol with new message types. No changes are proposed to existing message structures, or to their semantics.

With respect to backwards compatibility, both clients and servers need to be considered.

Servers

Servers that have not yet implemented the generic IO interface should not be affected, provided that they have proper mechanisms in place for dealing with unknown message types. The messages defined in this REP should result in an error being returned, with no further action by the server.

Clients

Clients that have not implemented the IO interface should not receive any IO related messages, as the server should not send any without having been instructed to do so by the client. The generic industrial robot client [5] currently prints an error on reception of unknown message types, and this should be sufficient to deal with any spurious IO messages from servers.

Reference Implementation

A reference implementation of a generic IO client will be made available in the `industrial_core` package. IO servers for each of the supported controller types will need to be implemented by maintainers of the respective drivers.

Impact on Other Tools

The Wireshark Simple Message dissector plugin [10] will be updated to add support for the new messages defined in this REP.

References

[1] ROS-Industrial simple_message package, ROS Wiki, on-line, retrieved 27 April 2014 (http://wiki.ros.org/simple_message)

[2] MT Connect ROS bridge - Fanuc custom gripper IO, MTConnect Institute GitHub organisation, on-line, retrieved 27 April 2014 (https://github.com/mtconnect/ros_bridge/blob/master/mtconnect_example/mtconnect_cnc_robot_example/karel/gripper.kl)

[3] CloPeMa Motoman IO extension, on-line, retrieved 27 April 2014 (http://clopema.felk.cvut.cz/gitweb/?p=ros_industrial.git;a=blob;f=motoman/dx100/motoplus/motoros_lib/io_handler.cpp;h=4e2f746f52a98c1200a218cf49c2634710fa24b6;hb=HEAD)

[4] Industrial Core - Native Robot I/O issue, ROS-Industrial GitHub organisation, on-line, retrieved 27 April 2014 (https://github.com/ros-industrial/industrial_core/issues/27)

[5] (*1*, *2*) ROS-Industrial robot client, ROS Wiki, on-line, retrieved 27 April 2014 (http://wiki.ros.org/industrial_robot_client)

[6] ROS-Industrial REP0001 - Industrial Robot Controller Motion/Status Interface (Version 2), ROS-Industrial GitHub organisation, on-line, retrieved 27 April 2014 (https://github.com/ros-industrial/rep/blob/6a80979f3aa427c48e1309124b27a2326dcdc843/rep-I0001.rst)

[7] MT Connect ROS bridge, MTConnect Institute GitHub organisation, on-line, retrieved 27 April 2014 (https://github.com/mtconnect/ros_bridge)

[8] ROS Patterns, Conventions: Naming ROS Resources, ROS Wiki, on-line, retrieved 28 April 2014 (http://wiki.ros.org/ROS/Patterns/Conventions)

[9] ROS Names, ROS Wiki, on-line, retrieved 28 April 2014 (http://wiki.ros.org/Names)

[10] Wireshark Simple Message dissector, ROS-Industrial GitHub organisation, on-line, retrieved 27 April 2014 (https://github.com/ros-industrial/packet-simplemessage)

[11] ABB, Rapid Reference Manual

[12] Comau, PDL2, Programming Language Manual

[13] DENSO Robot, PAC Programmer's Manual, Program Design and Commands

[14] EPSON, RC+, SPEL Language Reference

[15] FANUC Robot Series, R-30iA, Handling Tool, Operator's Manual

[16] FANUC Robot series, R-30iA, KAREL Function, Operator's Manual

[17] KUKA Roboter, System Software, Operating and Programming Instructions for System Integrators

[18]	Stäubli, VAL3 Reference Manual
[19]	Universal Robots, The URScript Programming Language
[20]	Yaskawa, Motoman, Instructions for Inform Language
[21]	Yaskawa, Motoman, Motoplus Reference (API Function Specifications)

Copyright