ProviewR

OPEN SOURCE PROCESS CONTROL

ProviewR Internals

2016-09-22

Table of Contents

About this Guide

The *ProviewR Internals* is intended for ProviewR developers and persons who want to have an insight in the internal processes of ProviewR.

Introduction

The Proview Internals describes the function of a number of different processes.

- Communication between processes and nodes.
- Alarm and event handling.
- Real time database.
- Nethandler.
- PLC program.

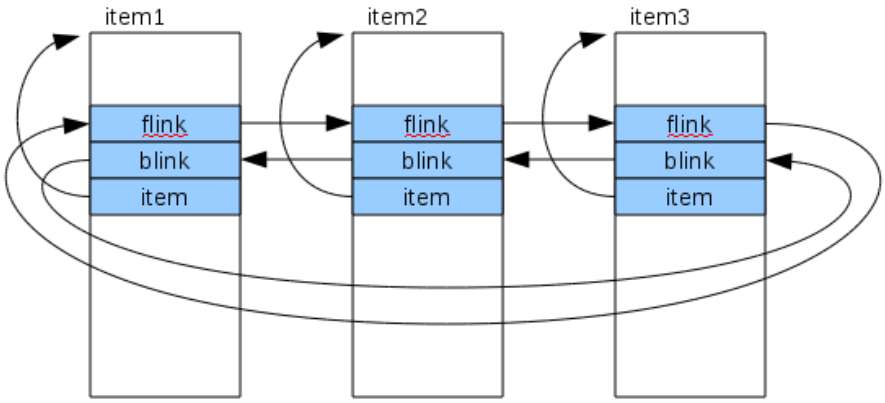
Common resources

Double linked lists

Lst is a module that handles double linked lists. Data structures that should be linked contains a list_sEntry struct with a forward pointer, a backward pointer, and a pointer to the start of the data structure. The forward and backward pointers points the the lst_sEntry structs in other data items.

```
struct lst_sEntry {
  lst_sEntry flink;  /* Forward pointer */
  lst_sEntry blink;  /* Backward pointer */
  void item;         /* Pointer to data structure */
}
```

The Lst list is a circular list so the last item always points to the first item.



A data structure can contain several lst_sEntry structs and thus be members of several lists.

There are a number of functions to add items and traverse the list. The first argument is a mutex for thread safe lists. It can be NULL if not used.

lst_Init(mx, entry, item)	Initialize an entry.
lst_InsertSucc()	Insert an item after the specified entry.
lst_InsertPred()	Insert an item before the specified entry.
lst_Succ()	Get the item after the specified entry.
lst_Pred()	Get the item before the specified entry.
lst_Remove()	Remove the specified entry.

A typical loop through all items in a list looks like this

```
lst_sEntry *entry, *start_entry;
void *item;

for (item = lst_Succ(NULL, start_entry, &entry);
     entry != start_entry;
     item = lst_Succ(NULL, entry, &entry)) {
```

```
  ...
}
```

Note that, as it is a circular list, the loop continues until the current entry equals the start_entry.

FIFO queue

Que is a FIFO queue that contains a Lst linked list, a condition and a mutex. With que_Put() and item is put on the queue and with que_Get() the oldest item is fetched. If the queue is empty when que_Get() is called, it will wait until an item is put on the queue, or until a specified timeout time has elapsed. As and Lst list is used internally, an item has to contain a lst_sEntry.

que_Init()	Initialize an queue.
que_Put()	Put an item on the queue.
que_Get()	Fetch the oldest item on the queue, or wait with timeout until an item in put on the queue.

Binary tree

A binary tree stores data structures with a key. A key compare function is supplied in the tree_CreateTable() call.

tree_CreateTable()	Create a table for a tree.
tree_DeleteTable()	Delete table.
tree_Insert()	Insert data for a specific key.
tree_Remove()	Remove data for a specific key.
tree_Find()	Find the data for a specific key.
tree_FindPredecessor()	Find the predecessor to the specified key.
tree_FindSuccessor()	Find the successor to the specified key.

Hash table

Communication

QCom monitor rt_qmon

rt_qmon handles all messages between nodes. The different types of messages are

- Alarms and events, This is communication between the event monitor and outunits, such as the alarm window in rt_xtt, or the event log.
- Nethandler communication. When processes requests information about objects or volumes in other nodes.
- Subscriptions. Messages with values that are updated cyclically, eg values displayed in a graph.
- Historical data. Values sent cyclically to the storage server, or data requested from the storage server.

Exported messages

Messages sent from one process to another makes a call to qcom_Put(). For communication with local processes, ie processes on the same node, the message is directly inserted into the receive queue. For processes on remote nodes they are inserted into the export queue and handled by the export thread in rt_qmon. The export thread divides the message into segments of usually 8192 bytes, and the segments are marked as first, middle or last segment. The segments

are placed in the link queue for the target node, and received by the link thread for this node. The link thread has a filter that removes old messages with the same message id. This assures that cyclic messages as subscriptions are not overflowing the thread queues as old one are removed when new with the corresponding data arrives. Segments that passes the filter are sent to the ethernet driver as an UDP message. The link thread waits for an ack for the last sent message before the next is sent. If no ack is received within the timeout time, the segment is resent, and the timeout time is doubled. It continues to be doubled until a specific time is exceeded, and the link will be disconnected. The ethernet driver is dividing the segments into packets of the RTO size, usually 1500 bytes.

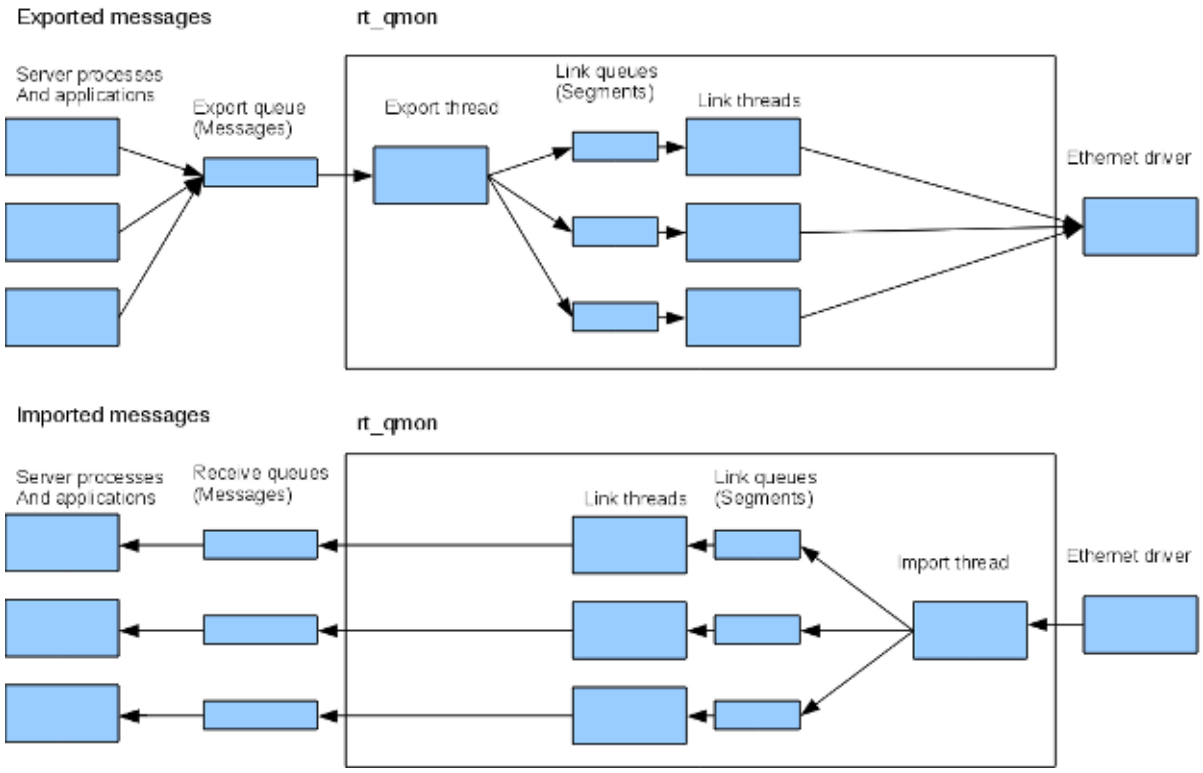
The initial timeout time is called MinResendTime and can be configured in the NodeConfig object. The default value is 50 ms. If no ack is received the MinResendTime will be doubled until the timeout time exceeds MaxResendTime, also configurable in the NodeConfig object. The default value is 10 s. Thus if no ack is received, the link is disconnected after 23 s.

Also the segment size can be configured in the NodeConfig object.

Imported messages

rt_qmon creates an UDP port to which QCom segments from other nodes are directed. The port number is 55000 + the QCom bus number.

The import thread receives the segments from the ethernet driver, and inserts them into the link queue for the sending node. This is actually the same as the link queue for exported messages, so the link queue contains both export and import segments. The segments are received by the link thread, and reassembled to a message. If it is an ack message, the corresponding export segment is removed, otherwise an ack is sent back, and the message is forwarded to the target queue, and from there received by the target process.



qcom_Put()

To send a qcom message qcom_Put() is called with the following data.

```
typedef struct {
  qcom_sQid     reply;     /* Reply queue */
  qcom_sType    type;      /* Message type b, and s */
  unsigned int  msg_id;    /* Message identity */
  unsigned int  prio;      /* Priority */
  unsigned int  allocate;  /* Data buffer should be allocated in the pool */
  unsigned int  size;      /* Data size */
  void          *data;      /* Data */
} qcom_sPut;
```

The data pointer points to the message. The message should be allocated in the QCom pool. If 'allocate' is 0, this should be done by the caller. If 'allocate' is 1, qcom_Put() will allocate and copy the message to the QCom pool.

size is the size in bytes of the message.

msg_id is set for cyclic messages as subscriptions and alarmstatus, and will cause old messages in the link queue with the same msg_id to be removed.

prio is not yet implemented.

Export queue

The export queue is created by rt_ini at runtime startup in qdb_BuildDb(). All messages addressed to remote nodes are inserted to this queue, and received by the export thread in rt_qmon.

Export thread

The export thread receives messages from the export queue. The messages are divided in segments with the configured segment size. The default size is 8192. The messages itself is still stored in the qcom pool and is kept there until the last segment is sent and acknowledged. For each segment, a

segment item is created with the segment size and a pointer to the segment data in the pool. The segment items for one message are linked in an Lst list, ie they are linked pointers to next and previous. Finally the segment items are inserted into the link queue for target node.

A broadcast message should be sent to all connected nodes, and segment items are created for each link and inserted into the link queues.

Segment size

A segment will be fragmented into ethernet packets by the ethernet driver. The size of these packets, RTO, are usually 1500 bytes. The segment size is configurable and a full segment with the default size, 8192 will result in 6 packets. If a packet is lost, the segment is not delivered and all packets has to be sent again. Thus for networks with few losses, larger segment sizes can be used, while for networks with frequent losses smaller segment sizes are preferred.

Link thread

The link thread reads the segment items from the link queue. This contains both items for outgoing message from the export thread, and for ingoing messages from the import thread.

Export segments

The link thread contains two linked lists, a send list and a window list. An export segment received from the link queue is first inserted into the send list, that contains segments that has not yet been sent. When the link is ready to send a new segment, the first segment in the send list is moved to the window list, and this segment is sent to the ethernet driver. Thus the window list contain segments that are send but yet not acked. A timeout time is set on the segment, and if this time expires and the segment is still present in the window list, it's sent again. The timeout time is fetched from MinResendTime and is 50 ms by default. If the first ack is not received within the timeout time, the time is doubled, and this continues until the MaxResendTime is reached (10 s by default). If still no ack it received the link is taken down. When an ack is received, the corresponding segment is freed and removed from the window list, and the first segment in the send list is moved to the window list. Currently the number of segments in the window list is limited to one. When the last segment in a message is freed, the message is also freed from the QCom pool.

Import segments

Import segments from the import thread are also received by the link thread from the link queue. If the segment is an ack, the corresponding export segment is found in the window list and freed. Otherwise if the segments are reassembled to a message. The message is allocated in the qcom pool when the first segment arrives and the content of this, and the following segments are copied to the message until the last segment arrives. When the message is complete it is put on the receiver queue stated in the message header.

Event and alarm handling

Event and alarm are triggered either from a set of supervising objects or from a function call in an application. The supervising objects are scanned by an agent for alarm condition, and when an alarm condition occurs, a detect flag is set in the object. The agent used depends on where the supervision object is located. Supervising objects in the plc code is scanned by the plc program, object under an I/O object is scanned by the io_comm process, and other object are scanned by the event monitor.

When the detect flag is set, messages are sent to outunits by the event monitor. An outunit can be an alarm or event window, or an event logging process.

Supervision objects

The supervision objects are

DSup	Supervision of a digital signal. A Dsup object can be placed in a plc program or in the plant hierarchy.
DSupComp	Supervision of a digital signal for components. The object is divided in a main and function object, where the main object is placed in the main object of the component, and the function object in the plc code.
ASup	Supervision of an analog signal. An ASup object can be placed in a plc program or in the plant hierarchy.
ASupComp	Supervision of analog singl for components.
SystemSup	Supervision of system functions. All SystemSup objects are placed in an array in the MessageHandler object.
CycleSup	Supervision of cyclic processes, eg plc threads.
LinkSup	Supervision of network links.

Agents

An agent scans supervision objects and detects when the alarm condition occurs and when it returns. It timestamps these events, and sets some flags in the object to indicate the current state for the event monitor. The communication with the event monitor is made with the following attributes in the supervision object.

AlarmCheck	Set by the event monitor when an object should be checked for alarm condition. Taken down by the agent when an alarm condition is detected and the delay time has elapsed.
DetectCheck	Also set by the event monitor when an object should be checked for alarm condition. Taken down by the agent when by the agent when an alarm condition is detected.
DetectSend	Set by the agent to indicate alarm condition. Taken down by the event monitor when the alarm is handled.
DetectTime	Time for alarm condition is detected.
ReturnCheck	Set by the agent to indicate that it should check for return event.Taken down by the agent when the return event has occurred.
ReturnSend	Set by the agent to message to the event monitor that a return has occurred. Reset

	by the event monitor when the return is handled.
ReturnTime	Time when the return is detected.
AlarmStatus	The two last bits indicates it the alarm is not returned (1), not acknowledge (2) or blocked (4).
AckTime	Time set by the event monitor when the alarm is acknowledged.

In short, the event monitor sets AlarmCheck and DetectCheck when it's ready to receive a new alarm. And the agent responds with DetectSend when the alarm is detected, and ReturnSend when the return is detected. When the alarm is return and acknowledged, AlarmCheck and DetectCheck is set again.

The agent is also counting down the time for the delay time.

The agent used depends on where the supervision object is located.

Plc program	The plc program scans the DSup, DSupComp, ASup and ASupComp objects with function objects in the plc code.
rt_io_comm	Scans supervision object placed under an I/O object.
rt_linksup	Handles NodeLinkSup objects.
rt_emon	Handles DSup and ASup objects not handled by any other agent, and SystemSup objects.

Alarms and events

Event monitor rt_emon

The event monitor identifies all supervision objects of class DSup, CompDSup, ASup, CompASup, CycleSup, NodeLinkSup or SystemSup and