

Timer

Inherits: **None**

A timer to emit messages periodically.

Description

Counts down a specified interval and emits a message to a Queue, User Event or custom message reference. Can be set to "Repeat" (default) or "One Shot" mode.

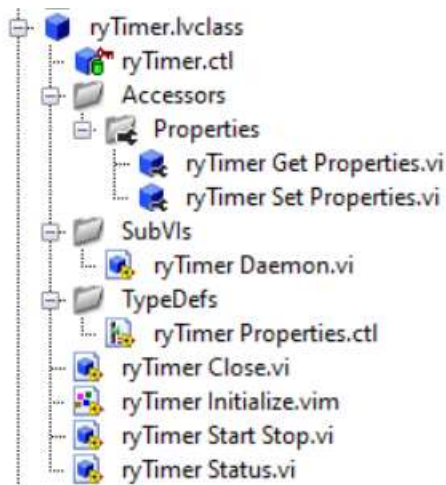
Timers are useful in situations where asynchronous messaging is needed to emit messages at periodic intervals. For example, querying an instrument's status periodically or returning telemetry at a consistent frequency.

A Queue's or Event's timeout case may be non-deterministic if the queue or user event is backlogged with messages, hence the need for a separate timer thread to ensure messages are emitted consistently.

Timer	1
Description	1
API	1
Examples	2
Queue (Repeated).....	2
Queue (One Shot).....	3
User Event (Repeated).....	4
Custom.....	4
Accessors	6
Properties.....	6
Methods	6
ryTimer Initialize.vim.....	6
ryTimer Start Stop.vi	7
ryTimer Status.vi	8
ryTimer Close.vi	8
Type Definitions	8
ryTimer Properties.ctl	8
SubVIs	9
ryTimer Daemon.vi	9
Callback	9

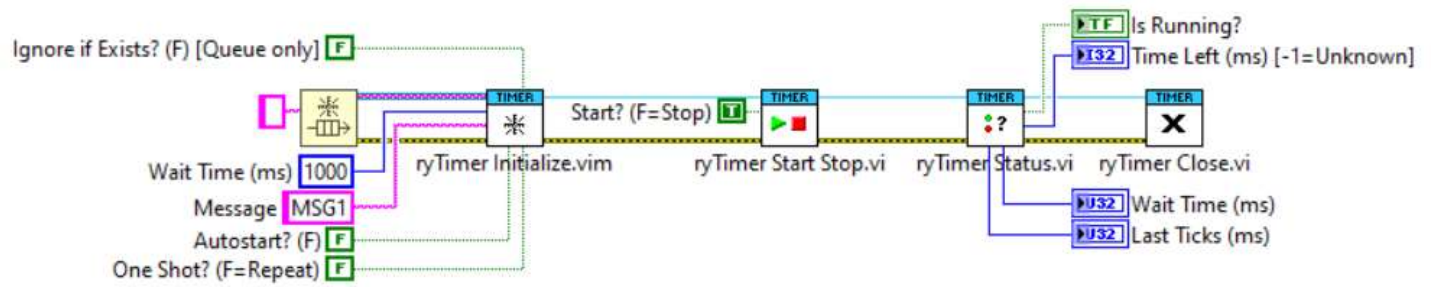
API

The timer class library is extremely compact with no external dependencies. It provides the flexibility to expand the built in references and messages types using callback mechanisms, essentially allowing for any reference to be treated as the emitter destination and any data structure to be emitted.



A timer only has 4 methods to initialize, start/stop, status & close the timer. The initialize malleable method accepts any destination reference such as **Queue**, **User Event**, notifier, etc.

String Queues & String User Events are supported out of the box and any other reference can be used for communication using the **Callback** mechanism (Refer to the **Callback** section below).



Initialize	vim	Accepts a Queue, User Event or custom reference, launches a timer daemon to periodically emit the message to the destination reference.
Start/Stop	vi	Starts or stops the timer. Optionally, sets the timer's wait time.
Status	vi	Return the status of the timer including the time left before the next emit.
Close	vi	Release and destroy the timer.

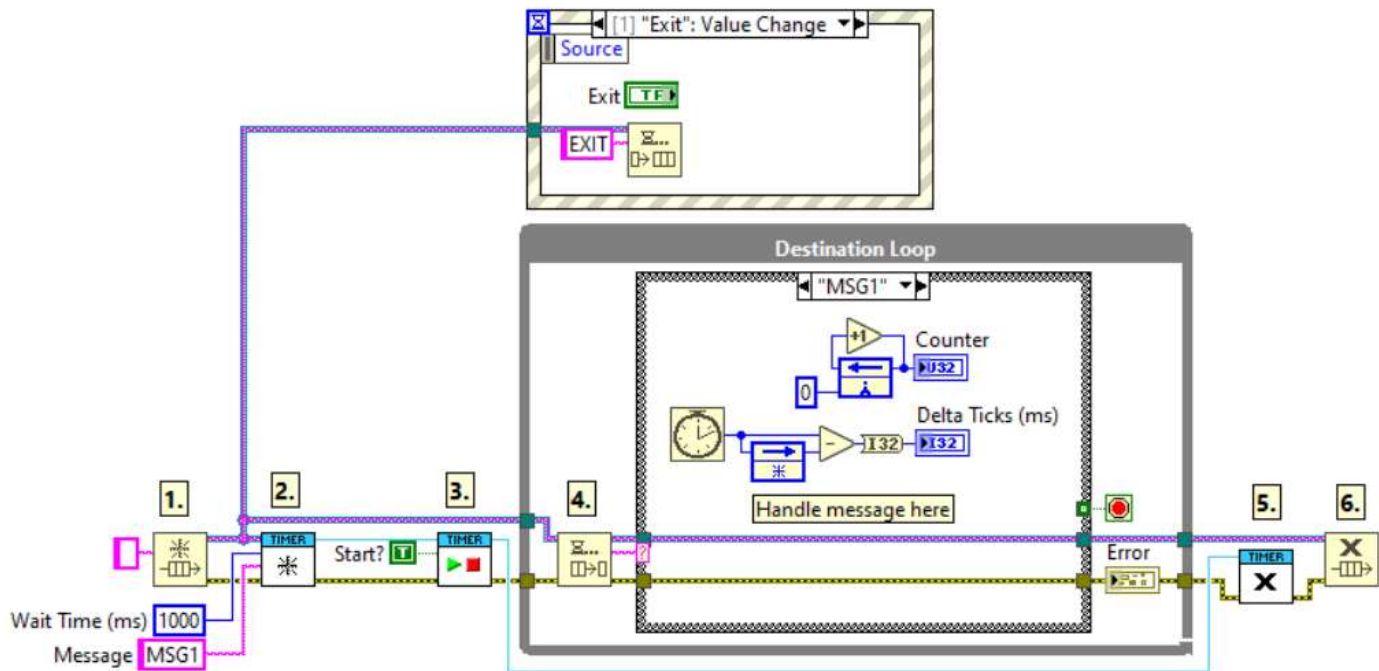
Examples

Queue (Repeated)

Enqueues a message repeatedly after each wait time. Used to create periodic messages, such as updating telemetry every few seconds.

By default a timer is set to repeat after each wait time, create a new timer and specify the **Wait Time (ms)** and **Message**. In this example, we use the **Start/Stop** method to manually start the timer.

*Note: **Autostart?** can be used in the initialize method to avoid calling the **Start/Stop** explicitly.*



1. Create a string **Destination Queue**

2. Create a new timer, specify the:

Destination Refnum - Queue reference where to enqueue the Message

Wait Time (ms) - Time in milliseconds to wait between each message enqueued

Message - The string message to enqueue

Note: By default Autostart?, One Shot? and Ignore if Exists? are False.

3. Start the timer

4. After each Wait Time (ms), the Message is enqueued to the Destination Queue

Note: More than one timer can be created to emit multiple messages at different asynchronous rates.

5. Release and destroy the timer

6. Release the Destination Queue

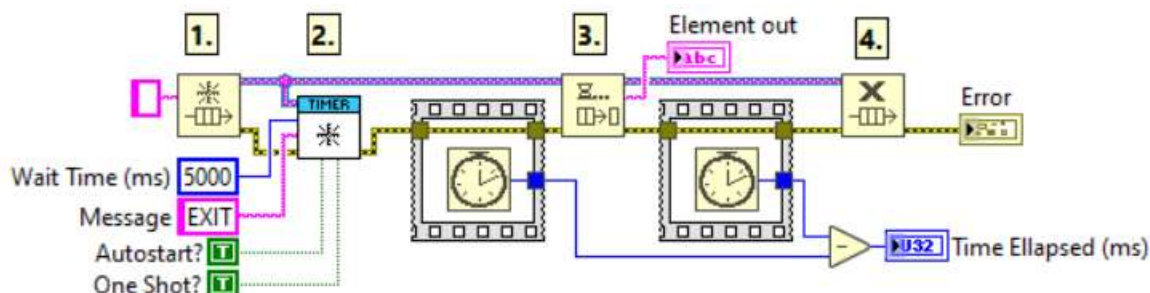
Note: The timer is destroyed automatically when the Destination Queue is released

Queue (One Shot)

Sometimes a delayed messaging is needed, the timer has the ability to do "One Shot" which will emit the message after the wait time then destroy itself.

To use One Shot, set **Autostart?** and **One Shot?** to **True**. This example will wait 5000ms then enqueue the "EXIT" message before destroying itself.

*Note: **Autostart?** must be set to **True** to automatically start the **One Shot**. If not, the **Start/Stop** method must be called to start the timer manually.*

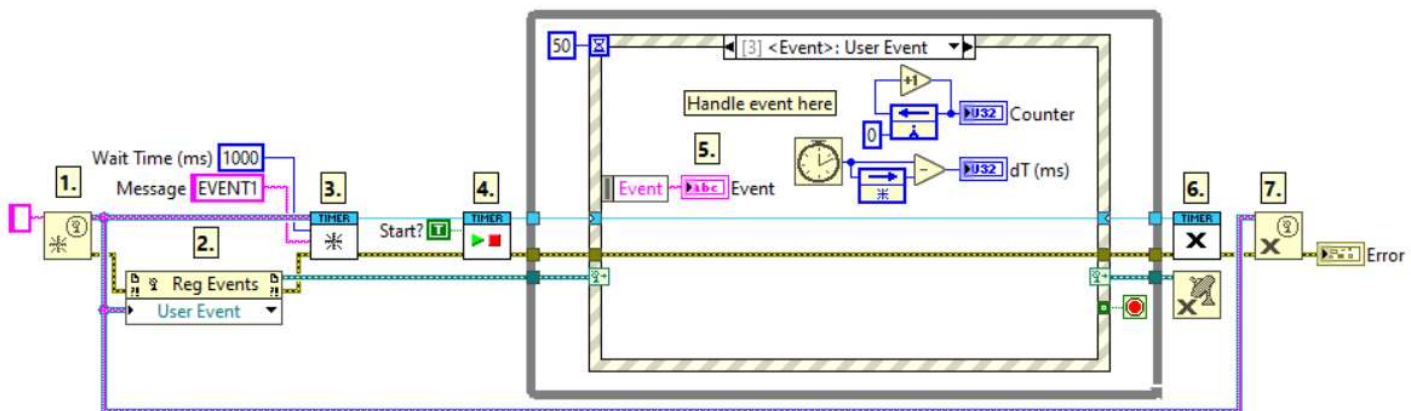


1. Create a string Destination Queue
2. Create a new timer, specify the: Destination Refnum - Queue reference to enqueue the Message Wait Time (ms) - Time in milliseconds to wait between each message enqueued Message - The string message to enqueue Autostart? - Automatically start the time. [Default = False] One Shot? - Automatically destroys the timer after the wait time ellapses and the message is sent.
3. After the Wait Time (ms) , the Message is enqueued to the Destination Queue
4. The timer is destroyed automatically when in One Shot mode. <i>Note: It is best practice to use the ryTimer Close.vi to manually dispose the timer but is not required.</i>

User Event (Repeated)

Similar to the Queue (Repeated) example, User Event timers are also supported.

*Note: It's important to **Register Events** prior to generating any event messages so that the events are not dropped.*

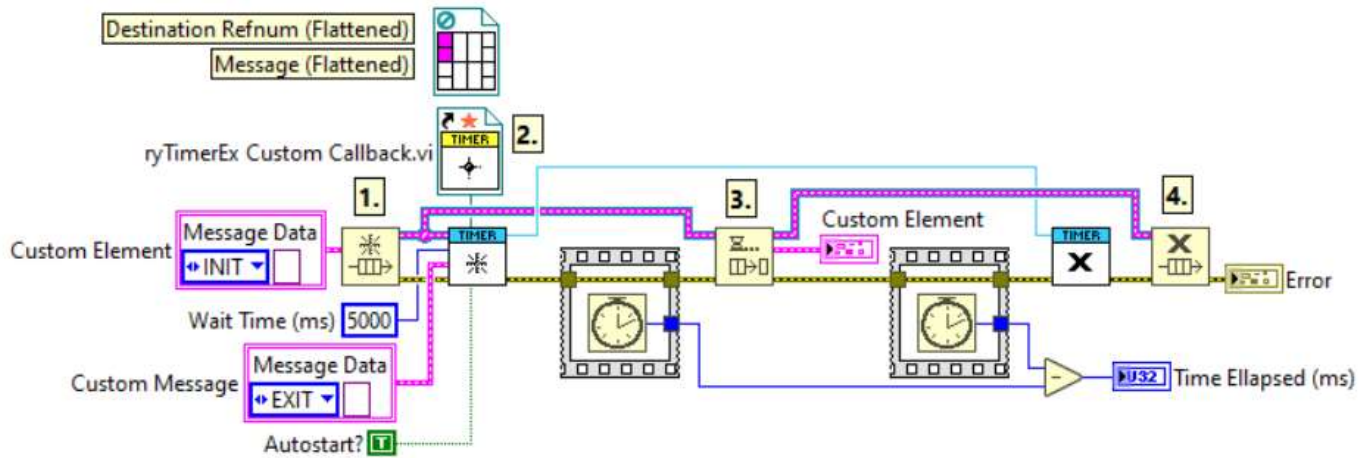


1. Create a string Destination User Event
2. Register the user event <i>Note: A user event must be registered before it will respond to an event, if not the event may be missed by the register.</i>
3. Create a new timer, specify the: Destination Refnum - User Event reference where to generate the event Message Wait Time (ms) - Time in milliseconds to wait between each message is generated Message - The string message to generate <i>Note: By default Autostart?, One Shot? and Ignore if Exists? are False.</i>
4. Start the timer
5. After each Wait Time (ms), the Message event is generated, handle the event in the event's handler case. <i>Note: More than one timer can be created to emit multiple messages at different asynchronous rates.</i>
6. Release and destroy the timer
7. Release the Destination User Event <i>Note: The timer is destroyed automatically when the Destination User Event is released</i>

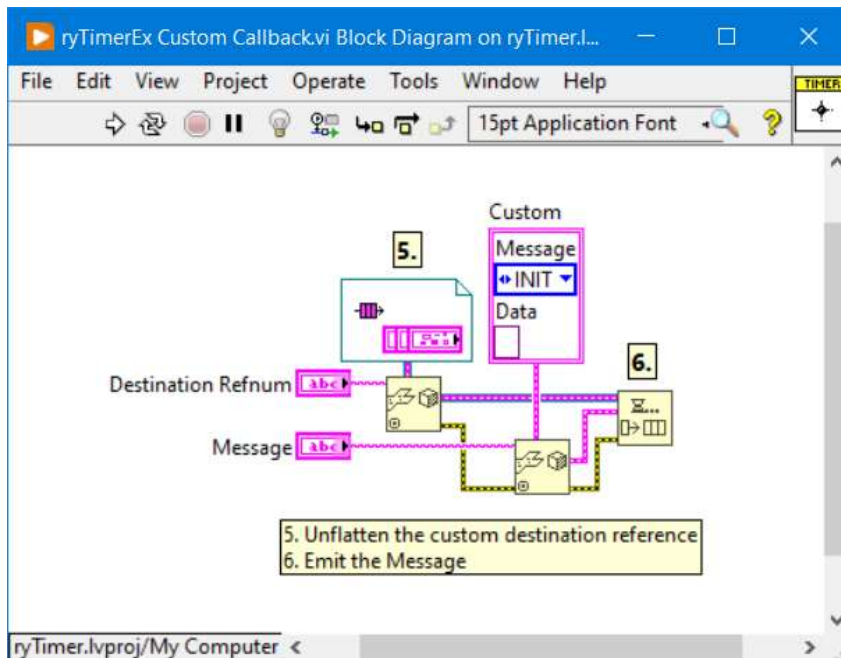
Custom

Out of the box, the timer supports string queues & user events. Custom queues, user events, notifiers, etc. and custom message data types can also be handled using Callback.

In this example, the timer is setup to handle a Enum + Variant Cluster Queue. The **Destination Refnum** and **Message** malleable inputs adapt to all data types that can be **Flattened to String**. The **Callback** VI unflattens the **Destination Refnum (Flattened string)** and **Message (Flattened string)** to emit the message to the destination refnum.



Where the **Callback** VI unflattens and enqueues the custom message data:



1. Create a custom data type **Destination Queue**

2. Create a new timer, specify the:

Custom Callback - VI to run after the time elapses, see the **Callback.vit** for usage.

Note: If the Custom Callback VI is NaN, the timer initialize with raise an error.

Destination Refnum - Custom queue reference which is flattened and passed to the callback VI.

Wait Time (ms) - Time in milliseconds to wait between each message enqueued

Message - The custom message which is flattened and passed to the callback VI to enqueue

Autostart? - Automatically start the time.

3. After each **Wait Time (ms)**, the **Callback** VI executes to enqueue the **Message** to the custom **Destination Queue**

4. The timer is destroyed automatically when the Destination Queue is released

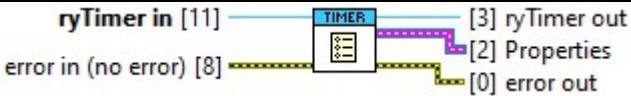
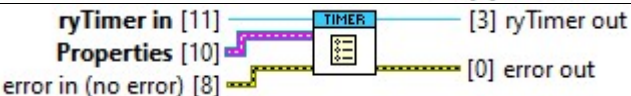
Note: It is best practice to use the ryTimer Close.vi to manually dispose the timer but is not required.

5. Unflatten the custom destination reference

6. Emit the Message

Accessors

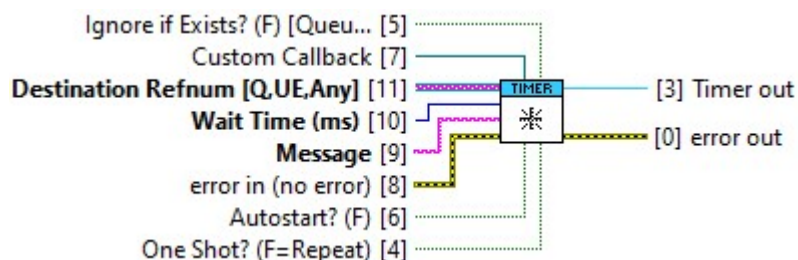
Properties

Getter	Public	
Setter	Public	

Get or set the Properties data value reference of the Type Def **Properties.ctl**.

Methods

ryTimer Initialize.vim



Ignore if Exists?

Type	Bool
Default	False

[For the built-in string queue only] If True, the Message is not emitted if it already exists on the queue.

Otherwise, if False the Message is always emitted regardless if the Message is already on the queue or not. This avoids potential backlogs of timer messages clogging up the queue.

Note: User Event and Custom timers ignore this option.

Custom Callback

Type	VI
Default	NaN

[For a custom destination refnum only] Create a **Static VI Reference** to a callback VI to handle a custom timer message. See the Custom example for usage.

Destination Refnum

Type	String Queue, String User Event, Any Refnum
Default	String Queue = NaN

This **Required** malleable input accepts any destination reference (i.e. Queue, User Event, Notifier, Occurrence, etc.) to handle custom timer messaging. By default String Queues and User Events are supported out of the box.

Wait Time (ms)

Type	U32
------	-----

Default	0
---------	---

This **Required** input sets the wait time in milliseconds between each emit. A value of zero pauses the timer.

*Note: The **Start/Stop** method can set the **Wait Time (ms)**.*

Message

Type	String
Default	""

This **Required** malleable input accepts any message data element to emit to the destination refnum after the wait time elapses. An error is raised if the message is empty.

Autostart?

Type	Bool
Default	False

Starts the timer when created. By default this is **False** and the **Start/Stop** method must be called explicitly to start the timer.

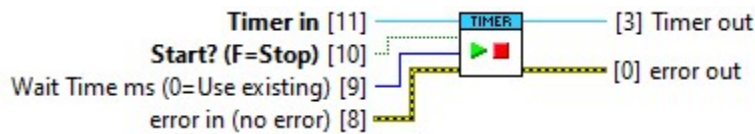
One Shot?

Type	Bool
Default	False

Sets the timer mode to "One Shot" which only emits the message one time before the timer is automatically destroyed. By default this is **False**, which repeats the timer after the wait time elapses.

*Note: The timer is automatically destroyed after a One Shot, there's no need to call the **Close** method.*

ryTimer Start Stop.vi



Start?

Type	Bool
Default	False

This **Required** input Starts (True) or Stops (False) the timer. The timer countdown is reset when start or stop is sent.

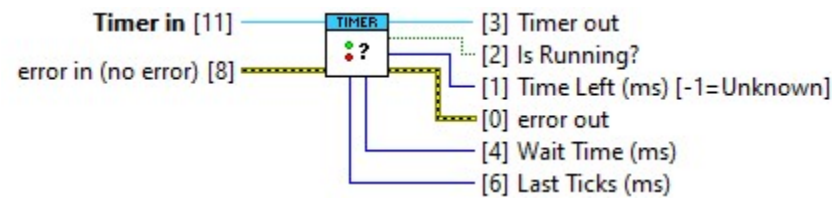
Wait Time (ms)

Type	U32
Default	0 (Use Existing)

This **Optional** input sets the wait time in milliseconds if the value is greater than one. Otherwise if zero, the wait time is not set and uses the existing **Wait Time (ms)** value.

*Note: The **Initialize** method initially sets the **Wait Time (ms)**. A value of zero pauses the timer.*

ryTimer Status.vi



Is Running?

Type	Bool
------	------

Returns True if the timer is running; False if paused.

Time Left (ms)

Type	I32
------	-----

Returns the time remaining in milliseconds until the next emit. If the timer is stopped, this returns -1.

Wait Time (ms)

Type	U32
------	-----

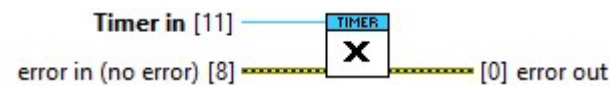
Returns the current wait time in milliseconds.

Last Ticks (ms)

Type	U32
------	-----

Returns the last internal tick count in milliseconds. Used to track emit timing characteristics.

ryTimer Close.vi



Closes and disposes the timer. Internal errors are ignored if the timer is already disposed.

Type Definitions

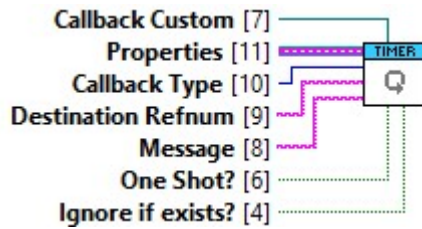
ryTimer Properties.ctl

```
Properties (typedef 'ryTimer.lvclass:ryTimer Properties.ctl'[non-strict])
Properties (cluster of 4 elements)
    Wait Time (ms) (unsigned long [32-bit integer (0 to 4,294,967,295)])
    Is Running (boolean (TRUE or FALSE))
    Last Ticks (unsigned long [32-bit integer (0 to 4,294,967,295)])
    Emit (Queue Refnum)
    Emit (boolean (TRUE or FALSE))
```

Wait Time (ms)	U32	Time in milliseconds to wait between emitting the message.
Is Running	Bool	Returns true if the timer is running and wait time is greater than zero.
Last Ticks	U32	Internal last tick count used to calculate the status time left (ms).
Emit	Queue	Single element queue used to throttle the timer daemon.

SubVIs

ryTimer Daemon.vi



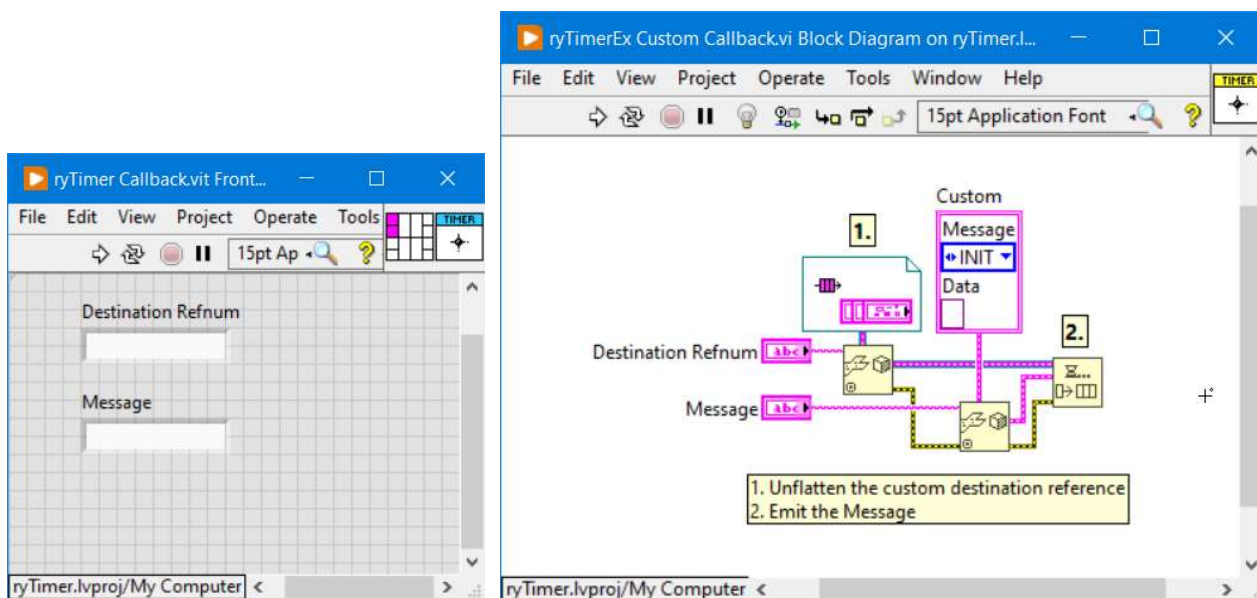
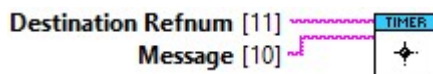
Timer daemon thread to periodically emit messages.

Callback

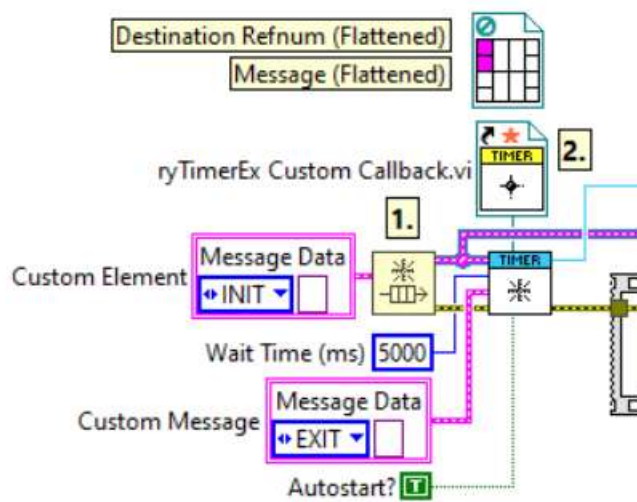
For custom **Destination Refnums** or **Message** data types, a callback VI can be used to emit the custom message to the custom reference. Create a new custom callback VI using the **Callback.vit** VI template. The callback VI flattens the Destination Refnum and Message to string, to be unflattened in the callback VI and emit the message (example shown below).

*Note: The callback VI must be **Shared** or **Preallocated Reentrant** since the **Call By Reference** node is used.*

*Note: The **Destination Refnum** and **Message** string connector pane terminals must be set to **Required**.*



Create a **Static VI Reference**, add the callback VI, make the static VI **Strictly Typed VI Reference** (adds red star). This keeps the VI in memory.



Note: The **Callback.vit** sets the VI Terminals to **Required**, Execution to **Preallocated Reentrant** and sets the VI **Custom Appearance** to **Uncheck the Show Horizontal Scrollbar**. This workaround flags the application builder to keep the front panel when compiling the Run-Time Application, ensuring the Static VI call will work properly in different environments.