AVRToolsPlus

Generated by Doxygen 1.8.13

Contents

Chapter 1

AVRToolsPlus: A library of higher-level tools for the AVR ATmega328 and ATmega2560 Microcontrollers

1.1 Introduction

The AVRToolsPlus library provides a collection of functionality for the AVR family of 8-bit microcontrollers. Unlike the low-level functionality found in the AVRTools library (which very directly interfaces with AVR hardware subsystems), the functionality in the AVRToolsPlus library is more high-level and does not relate directly to hardware subsystems. In fact, the modules in AVRToolPlus are sufficiently generic that they often don't even make use of AVRTools functionality. Nevertheless, the code in AVRToolsPlus is specifically intended for use on AVR 8-bit microcontrollers and the implementation often relies on specific characteristics of this family of microcontrollers.

For this reason, rather add these modules to the AVRTools library, I decided to package them in a separate library.

1.2 Overview

The AVRToolsPlus library includes a collection tools that provide higher-level functionality. These tools are designed for the AVR family of 8-bit microcontrollers, and while the implementation of AVRToolsPlus modules depends on characteristics of the AVR 8-bit architecture, these modules are not intimately tied to any specific AVR hardware subsystems. For this reason, AVRToolsPlus modules should run on any of AVR's 8-bit microcontrollers (although I have only tested them on ATmega328 and ATmega2560).

The tools in AVRToolsPlus are organized as a collection of independent modules. These modules are:

- EventManager module

The AVRToolsPlus modules do not depend on any of the AVRTools modules, but are fully interoperable with AVRTools.

Documentation

Detailed documentation is provided by this PDF document located in the repository, or online in HTML form.

Feedback

If you find a bug or if you would like a specific feature, please report it at:

https://github.com/igormiktor/AVRToolsPlus/issues

If you would like to hack on this project, don't hesitate to fork it on GitHub. If you would like me to incorporate changes you made, please send me a Pull Request.

Chapter 2

EventManager

Using an event-driven design is a common way to code AVR projects that interact with the environment around them. With EventManager you register functions that "listen" for particular events. When things happen you "post" events to EventManager. You then periodically call an EventManager function which processes events by dispatching them so that he appropriate listeners are called.

EventManger is interrupt safe, so that you can post events from interrupt handlers. The corresponding listeners will be called from outside the interrupt handler in your main thread of execution when you call the EventManager function to process events.

In keeping with the limited resources of an AVR system, EventManager is light-weight. There is no dynamic memory allocation. Event queuing is very fast, so you can be comfortable queuing events from interrupt handlers. To keep the footprint minimal, the event queue and the listener list (also known as the dispatch table) are both small (although you can make them bigger if needed).

EventManager supports both high priority and low priority events, and will process high priority events ahead of any low priority events.

EventManager is implemented as a set of functions contained with the namespace EventManager. Implementing EventManager this was instead of as a class removes the need to create and share an EventManager object across your code. However, it does limit your code to a single EventManager and therefore a single set of event queues.

2.1 Usage

Using EventManager is straightforward. You include EventManager.h in files that access the EventManager and link against the file `EventManager.cpp`. You register listener functions using EventManager::addListener(), you post events using EventManager::queueEvent(), and you process events by calling EventManager::processEvent().

The following sections explain this in more detail.

2.2 Events

EventManager events consist of an event code and an event parameter. Both of these are integer values. The event code identifies the type of event. For convenience, EventManager.h provides a set of constants you can use to identify events:

```
EventManager::kEventKeyPress
EventManager::kEventKeyRelease
EventManager::kEventChar
EventManager::kEventTime
EventManager::kEventTimer0
EventManager::kEventTimer1
EventManager::kEventTimer2
EventManager::kEventTimer3
EventManager::kEventAnalog0
EventManager::kEventAnalog1
EventManager::kEventAnalog2
EventManager::kEventAnalog3
EventManager::kEventAnalog4
EventManager::kEventAnalog5
EventManager::kEventMenu0
EventManager::kEventMenu1
EventManager::kEventMenu2
EventManager::kEventMenu3
EventManager::kEventMenu4
EventManager::kEventMenu5
EventManager::kEventMenu6
EventManager::kEventMenu7
EventManager::kEventMenu8
EventManager::kEventMenu9
EventManager::kEventSerial
EventManager::kEventPaint
EventManager::kEventUser0
EventManager::kEventUser1
EventManager::kEventUser2
EventManager::kEventUser3
EventManager::kEventUser4
EventManager::kEventUser5
EventManager::kEventUser6
EventManager::kEventUser7
EventManager::kEventUser8
EventManager::kEventUser9
```

These are purely for your convenience; EventManager only uses the numerical value to match events to listeners, so you are free to use any event codes you wish.

The event parameter is also whatever you want it to be. EventManager simply passes the event parameter to every listener function that is associated with that event code. For example, for a key press event the event parameter could be the corresponding key code. For an analog event it could be the value read from that analog pin or a pin number.

You post events using the EventManager::queueEvent() function, like this:

```
EventManager::queueEvent( EventManager::kEventUser0, 1234 );
```

The EventManager::queueEvent() function is lightweight and interrupt safe, so you can call it from inside an interrupt handler.

By default the event queue holds 8 events, but you can make the event queue any size you want by defining the macro EVENTMANAGER_EVENT_QUEUE_SIZE to whatever value you desire (see `Increasing Event Queue Size` below).

2.3 Listeners

Listeners are functions of type

```
typedef void ( *EventListener )( int eventCode, int eventParam );
```

You add listeners using the EventManager::addListener() function, like this:

```
void myListener( int eventCode, int eventParam )
{
    // Do something with the event
}

void setup()
{
    EVentManager::addListener( EventManager::kEventUser0, myListener );

    // Do more set up
}
```

Note

Do *not* add listeners from within an interrupt routine.

By default the list of listeners holds 8 listeners, but you can make the list any size you want by defining the macro EVENTMANAGER_DISPATCH_TABLE_SIZE to whatever value you desire (see `Increasing Listener List Size` below).

2.4 Processing Events

To process events in the event queue and dispatch them to listeners you call EventManager::processEvent() from a top-level event handling loop:

```
void loop()
{
    // Do stuff that might generate events

    // Process events
    EventManager::processEvent();
}
```

This call processes one event from the event queue every time it is called. Usually EventManager::processEvent() is called once in the inside of some main event loop so that one event is handled every time through the loop. This is usually more than adequate to keep up with incoming events. Events are normally processed in a first-in, first-out fashion (but see the section on `Event Priority` below).

2.5 Advanced Details

The remaining sections provide additional details about Event Manager that allowed more tailor and flexible usage.

2.5.1 Event Priority

EventManager recognizes high and low priority events. You can specify the priority when you queue the event. By default, events are considered low priority. You indicate an event is high priority by passing an additional constant to EventManager::queueEvent(), like so:

```
EventManager::queueEvent( EventManager::kEventUser0, 1234,
      EventManager::kHighPriority );
```

The difference between high and low priority events is that EventManager::processEvent() will process a high priority event ahead of any low priority events. In effect, high priority events jump to the front of the queue (multiple high priority events are processed first-in, first-out, but all of them are processed before any low priority events).

Note

If high priority events are queued faster than low priority events, EventManager may never get to processing any of the low priority events. So use high priority events judiciously.

2.5.2 Interrupt Safety

EventManager is interrupt safe, so that you can queue events both from within interrupt handlers and also from normal functions without having to worry about queue corruption. This safety is achieved by globally disabling interrupts while certain small snippets of code are executing.

2.5.3 Processing All Events

Normally calling EventManager::processEvent() once every time through a top-level event processing loop is more than adequate to service incoming events. However, there may be times when you want to process all the events in the queue. For this purpose you can call EventManager::processAllEvents(). Note that if you call this function at the same time that a series of events are being rapidly added to the queue asynchronously (for example, via interrupt handlers), the EventManager::processAllEvents() function might not return until the series of additions to the event queue stops.

2.5.4 Increasing Event Queue Size

Define the macro `EVENTMANAGER_EVENT_QUEUE_SIZE` to whatever size you need at compile time by passing it to the compiler on the command line using something like: `-DEVENTMANAGER_EVENT_QUEUE_SIZE=32`

The event queue requires `4 * sizeof(int) = 8` bytes for each unit of size. There is a factor of 4 (instead of 2) because internally EventManager maintains two separate queues: a high-priority queue and a low-priority queue.

2.5.5 Increasing Listener List Size

Define the macro `EVENTMANAGER_LISTENER_LIST_SIZE` to whatever size you need at compile time by passing it to the compiler on the command line using something like: : `-DEVENTMANAGER_LISTENER_LIST_SIZE=32`

The listener list requires `sizeof(*f()) + sizeof(int) + sizeof(boolean) = 5` bytes for each unit of size.

2.5.6 Additional Features

There are various functions for managing the listeners:

- You can remove listeners using EventManager::removeListener()
- Disable and enable specific listeners using EventManager::enableListener()
- Set a default listener using EventManager::setDefaultListener(); the default listener will handle any events not handled by other listeners
- Manipulate the default listener just like any other listener using EventManager::removeDefaultListener() and EventManager::enableDefaultListener()
- Check if the listener list is full using EventManager::isListenerListFull()
- Check if the listener list is empty using EventManager::isListenerListEmpty()

There are various class functions that provide information about the event queue:

- Check if the event queue is full using EventManager::isEventQueueFull()
- Check if the event queue is empty using EventManager::isEventQueueEmpty()
- See how many events are in the event queue using EventManager::getNumEventsInQueue()

For details on these functions you should review EventManager.h documentation.

2.5.7 Credits

EventManager was inspired by the Arduino Event System by mromani@ottotecnica.com of OTTOTECNICA Italy.

Chapter 3

Namespace Index

3.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 EventManager Namespace Reference

This namespace bundles the Event Manager functionality. It provides logical cohesion for functions implement the Event Manager and prevents namespace collisions.

Typedefs

- typedef void(∗ EventListener) (int eventCode, int eventParam)

  Type for an event listener (a.k.a. callback) function.

Enumerations

- enum GenericEvents

  This enum provides common event names, purely for user convenience.
- enum EventPriority

  EventManager recognizes two kinds of events. By default, events are are queued as low priority, but these constants can be used to explicitly set the priority when queueing events.

Functions

- bool addListener (int eventCode, EventListener listener)

  Add an (event, listener) pair listener to the dispatch table.
- bool removeListener (int eventCode, EventListener listener)

  Remove this (event, listener) pair from the dispatch table. Other listener pairs with the same function or event code will not be affected.
- int removeListener (EventListener listener)

  Remove all occurrances of a listener from the dispatch table, regardless of the event code. returns number removed.
- bool enableListener (int eventCode, EventListener listener, bool enable)

  Enable or disable an (event, listener) pair entry in the dispatch table.

- bool isListenerEnabled (int eventCode, EventListener listener)

  Obtain the the current enabled/disabled state of an (eventCode, listener) pair.
- bool setDefaultListener (EventListener listener)

  Set a default listener. The default listener is a callback function that is called when an event with no listener is processed.
- void removeDefaultListener ()

  Remvoes the default listener. The default listener is a callback function that is called when an event with no listener is processed.
- void enableDefaultListener (bool enable)

  Enable or disable the default listener. The default listener is a callback function that is called when an event with no listener is processed.
- bool isListenerListEmpty ()

  Check if the listener list (a.k.a., dispatch table) is empty.
- bool isListenerListFull ()

  Check if the listener list (a.k.a., dispatch table) is full.
- int numListeners ()

  Get the number of listeners in the dispatch table.
- bool isEventQueueEmpty (EventPriority pri=kLowPriority)

  Check if the event queue is empty.
- bool isEventQueueFull (EventPriority pri=kLowPriority)

  Check if the event queue is full.
- int getNumEventsInQueue (EventPriority pri=kLowPriority)

  Get the number of events in the event queue.
- bool queueEvent (int eventCode, int eventParam, EventPriority pri=kLowPriority)

  Tries to add an event into the event queue.
- int processEvent ()

  Processes one event from the event queue and dispatches it to the corresponding listeners stored in the dispatch table.
- int processAllEvents ()

  Processes all the events in the event queues and dispatches them to the corresponding listeners stored in the dispatch table.

5.1.1 Detailed Description

This namespace bundles the Event Manager functionality. It provides logical cohesion for functions implement the Event Manager and prevents namespace collisions.

5.1.2 Enumeration Type Documentation

5.1.2.1 EventPriority

```
enum EventManager::EventPriority
```

EventManager recognizes two kinds of events. By default, events are are queued as low priority, but these constants can be used to explicitly set the priority when queueing events.

Note

High priority events are always handled before any low priority events.

5.1.3 Function Documentation

5.1.3.1 addListener()

```
bool EventManager::addListener (
            int eventCode,
            EventListener listener )
```

Add an (event, listener) pair listener to the dispatch table.

- `eventCode` the event code this listener listens for.
- `listener` the listener to be called when there is an event with this eventCode.

Returns

True if (the event, listener) pair is successfully installed in the dispatch table, false otherwise (e.g. the dispatch table is full).

5.1.3.2 enableDefaultListener()

```
void EventManager::enableDefaultListener (
            bool enable )
```

Enable or disable the default listener. The default listener is a callback function that is called when an event with no listener is processed.

- `enable` pass true to enable the default listener, false to disable it.

5.1.3.3 enableListener()

```
bool EventManager::enableListener (
            int eventCode,
            EventListener listener,
            bool enable )
```

Enable or disable an (event, listener) pair entry in the dispatch table.

- `eventCode` the event code of the (event, listener) pair to be enabled or disabled.
- `listener` the listener of the (event, listener) pair to be enabled or disabled.
- `enable` pass true to enable the (event, listener) pair, false to disable it.

Returns

True if the (event, listener) pair was successfully enabled or disabled, false if the (event, listener) pair was not found.

5.1.3.4 getNumEventsInQueue()

```
int EventManager::getNumEventsInQueue (
            EventPriority pri = kLowPriority )
```

Get the number of events in the event queue.

- `pri` the desired event queue: kLowPriority or kHighPriority. Defaults to kLowPriority.

Returns

The number of events in the specified event queue.

5.1.3.5 isEventQueueEmpty()

```
bool EventManager::isEventQueueEmpty (
            EventPriority pri = kLowPriority )
```

Check if the event queue is empty.

- `pri` the desired event queue: kLowPriority or kHighPriority. Defaults to kLowPriority.

Returns

True if the specified event queue is empty.

5.1.3.6 isEventQueueFull()

```
bool EventManager::isEventQueueFull (
            EventPriority pri = kLowPriority )
```

Check if the event queue is full.

- `pri` the desired event queue: kLowPriority or kHighPriority. Defaults to kLowPriority.

Returns

True if the specified event queue is full.

5.1.3.7 isListenerEnabled()

```
bool EventManager::isListenerEnabled (
            int eventCode,
            EventListener listener )
```

Obtain the the current enabled/disabled state of an (eventCode, listener) pair.

- `eventCode` the event code of the (event, listener) pair.
- `listener` the listener of the (event, listener) pair.

Returns

The current enabled/disabled state of the (eventCode, listener) pair.

5.1.3.8 isListenerListEmpty()

```
bool EventManager::isListenerListEmpty ( )
```

Check if the listener list (a.k.a., dispatch table) is empty.

Returns

True if the listener list (dispatch table) is empty, false if not.

5.1.3.9 isListenerListFull()

```
bool EventManager::isListenerListFull ( )
```

Check if the listener list (a.k.a., dispatch table) is full.

Returns

True if the listener list (dispatch table) is full, false if not.

5.1.3.10 numListeners()

```
int EventManager::numListeners ( )
```

Get the number of listeners in the dispatch table.

Returns

The number of entries in the listener list (a.k.a., dispatch table).

5.1.3.11 processAllEvents()

```
int EventManager::processAllEvents ( )
```

Processes *all* the events in the event queues and dispatches them to the corresponding listeners stored in the dispatch table.

Events are taken preferentially from the high priority queue. If the high priority queue is empty, then events are taken from the low prioirty queue.

All listeners associated with the event that are enabled will be called. Disabled listeners are not called.

The event processed is removed from the event queue (even if there is no listener to handle it).

Note

If interrupts or other asynchronous processes are adding events as they are being processed, this function might not return for a long time. If events are added as quickly as this function processes them, this function will never return. .

Returns

The number of event handlers called.

5.1.3.12 processEvent()

```
int EventManager::processEvent ( )
```

Processes one event from the event queue and dispatches it to the corresponding listeners stored in the dispatch table.

Events are taken preferentially from the high priority queue. If the high priority queue is empty, then events are taken from the low prioirty queue.

All listeners associated with the event that are enabled will be called. Disabled listeners are not called.

The event processed is removed from the event queue (even if there is no listener to handle it).

Note

This function should be called regularly to keep the event queues from getting full. This is usually done by calling it inside an event processing loop.

Returns

The number of event handlers called.

5.1.3.13 queueEvent()

```
bool EventManager::queueEvent (
            int eventCode,
            int eventParam,
            EventPriority pri = kLowPriority )
```

Tries to add an event into the event queue.

- `eventCode` identifies the event to be added.
- `eventParam` an integer parameter associated with this event.
- `pri` specifies which queue gets the event: kLowPriority or kHighPriority. Defaults to kLowPriority.

Returns

True if successful; false if the queue is full and the event cannot be added.

5.1.3.14 removeListener() [1/2]

```
bool EventManager::removeListener (
            int eventCode,
            EventListener listener )
```

Remove this (event, listener) pair from the dispatch table. Other listener pairs with the same function or event code will not be affected.

- `eventCode` the event code of the (event, listener) pair to be removed.
- `listener` the listener of the (event, listener) pair to be removed.

Returns

True if the (event, listener) pair is successfully removed, false otherwise.

5.1.3.15 removeListener() [2/2]

```
int EventManager::removeListener (
            EventListener listener )
```

Remove all occurrances of a listener from the dispatch table, regardless of the event code. returns number removed.

This function is useful when one listener handles many different events.

- `listener` the listener to be removed.

Returns

The number of entries removed from the dispatch table.

5.1.3.16 setDefaultListener()

```
bool EventManager::setDefaultListener (
            EventListener listener )
```

Set a default listener. The default listener is a callback function that is called when an event with no listener is processed.

- `listener` the listener to be set as the default listener.

Returns

True if the default listener is successfully installed, false if `listener` is null.

Chapter 6

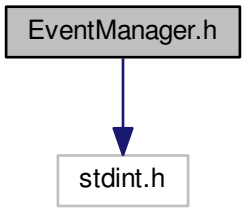
File Documentation

6.1 EventManager.h File Reference

This file provides an Event Manager using functional interface under the namespace EventManager.

```
#include <stdint.h>
```

Include dependency graph for EventManager.h:



Namespaces

- EventManager

  This namespace bundles the Event Manager functionality. It provides logical cohesion for functions implement the Event Manager and prevents namespace collisions.

Typedefs

- typedef void(∗ EventManager::EventListener) (int eventCode, int eventParam)

  Type for an event listener (a.k.a. callback) function.

Enumerations

- enum EventManager::GenericEvents

  This enum provides common event names, purely for user convenience.
- enum EventManager::EventPriority

  EventManager recognizes two kinds of events. By default, events are are queued as low priority, but these constants can be used to explicitly set the priority when queueing events.

Functions

- bool EventManager::addListener (int eventCode, EventListener listener)

  Add an (event, listener) pair listener to the dispatch table.
- bool EventManager::removeListener (int eventCode, EventListener listener)

  Remove this (event, listener) pair from the dispatch table. Other listener pairs with the same function or event code will not be affected.
- int EventManager::removeListener (EventListener listener)

  Remove all occurrances of a listener from the dispatch table, regardless of the event code. returns number removed.
- bool EventManager::enableListener (int eventCode, EventListener listener, bool enable)

  Enable or disable an (event, listener) pair entry in the dispatch table.
- bool EventManager::isListenerEnabled (int eventCode, EventListener listener)

  Obtain the the current enabled/disabled state of an (eventCode, listener) pair.
- bool EventManager::setDefaultListener (EventListener listener)

  Set a default listener. The default listener is a callback function that is called when an event with no listener is processed.
- void EventManager::removeDefaultListener ()

  Remvoes the default listener. The default listener is a callback function that is called when an event with no listener is processed.
- void EventManager::enableDefaultListener (bool enable)

  Enable or disable the default listener. The default listener is a callback function that is called when an event with no listener is processed.
- bool EventManager::isListenerListEmpty ()

  Check if the listener list (a.k.a., dispatch table) is empty.
- bool EventManager::isListenerListFull ()

  Check if the listener list (a.k.a., dispatch table) is full.
- int EventManager::numListeners ()

  Get the number of listeners in the dispatch table.
- bool EventManager::isEventQueueEmpty (EventPriority pri=kLowPriority)

  Check if the event queue is empty.
- bool EventManager::isEventQueueFull (EventPriority pri=kLowPriority)

  Check if the event queue is full.
- int EventManager::getNumEventsInQueue (EventPriority pri=kLowPriority)

  Get the number of events in the event queue.
- bool EventManager::queueEvent (int eventCode, int eventParam, EventPriority pri=kLowPriority)

  Tries to add an event into the event queue.
- int EventManager::processEvent ()

  Processes one event from the event queue and dispatches it to the corresponding listeners stored in the dispatch table.
- int EventManager::processAllEvents ()

  Processes all the events in the event queues and dispatches them to the corresponding listeners stored in the dispatch table.

6.1.1 Detailed Description

This file provides an Event Manager using functional interface under the namespace EventManager.

To use these functions, include EventManager.h and link against EventManager.cpp.

The event queue and and listener list are arrays of fixed size. The size of both can be set at compile time.

The default size of the dispatch table is 8 (event, listener) pairs; you can change the default at compile time by defining the macro `EVENTMANAGER_DISPATCH_TABLE_SIZE` prior to including the file EventManager.h, each time it is included. Define it using a compiler option (e.g., `-DEVENTMANAGER_DISPATCH_TABLE_SIZE=32`) to ensure it is consistently defined throughout your project.

The default size of the event queues is 8 events (note there are two queues, one for routine priority events), the other for high priority events. You can change the default size of both of the event queues at compile time by defining the macro `EVENTMANAGER_EVENT_QUEUE_SIZE` prior to including the file EventManager.h, each time it is included. Define it using a compiler option (e.g., `-DEVENTMANAGER_EVENT_QUEUE_SIZE=32`) to ensure it is consistently defined throughout your project.

Index