ARDUINO Microcontroller External Interrupt Handling

User Guide

Scalable, Asynchronous Processing

of External Interrupts

{A generic method for the handling of multiple external interrupts}

Change Control Record

Date	Version	Remarks	Author
03/03/2020	v2.03.1-D	Based on version 2.03 of the framework. Initial version for review	R D Bentley

Contents

A Note About Terminology

1. The ARDUINO development environment uses the term ‘sketch’ to refer to the source C++ code comprising a program. In this guide, rather than ‘sketch’, the terms ‘solution’ and ‘application’ are used as these terms provide a higher level understanding of the intended outcome of an ARDUINO sketch.
2. In the context of this guide, the term ‘end user’ refers specifically to designers and developers of ARDUINO solutions and applications.

Introduction

This document presents an asynchronous method for a scalable framework for applications requiring the use of external interrupts for ARDUINO microcontrollers (or compatible alternatives, e.g. Elegoo).

The method will allow external interrupts to be processed in their order of priority (when triggered concurrently and in real time), but at such time that the main processing code dictates (asynchronously). That is, the method allows for external interrupts to be captured, as triggered, but actioned outside and away from their associated Interrupt Service Routines (ISRs).

The solution has been developed around the ARDUINO MEGA 2560 microcontroller, but it may be modified for any board in a straight forward way for any number of external interrupts up to the maximum limit set by the specific microcontroller and this framework (see Configuration).

The MEGA 2560 supports up to six external interrupts that are readily accessible to end user developers. These interrupts are mapped to digital input pins; these assignments can be seen at Appendix A. However, the method is easily scaled to use one, two, three,...,or all six external interrupts concurrently, making it an ideal framework on which to base solutions requiring multiple external interrupt processes and processing. For microcontrollers that support more than six interrupts an extension to the underlying framework would be needed. However, it is considered that a limit of six interrupts should not be a limiting factor for most applications likely to be developed. If more are required then the method could be extended by increasing the number of generic ISRs and associated data support structures.

Asynchronous processing is achieved through the design and implementation of an interrupt queue management concept which provides the means to separate the event of an interrupt from its ultimate processing.

When an external interrupt is triggered, it is presented the nominated (generic) ISR mapped and established within the setup() process. ISRs will then do little more than insert the interrupt event into the interrupt queue with no further process taking place at this point. However, to avoid ‘bounce’ from spurious/dirty interrupts, each ISR incorporates a debounce mechanism which is variable through configuration parameters. Interrupt ‘bounce’ values may defined differently for each mapped external interrupt, as necessary.

If several external interrupts are triggered at the same time then the ARDUINO microcontroller will ‘fire’ their respective ISRs in the defined priority order as determined by the microcontroller specifications (see Appendix A for the MEGA 2560 priorities).

The interrupt queue is operated and managed on a first in, first out basis (FIFO). The approach therefore preserves the order in which external interrupts are triggered in the real world.

Following configuration, to meet a specific solution, end users have the ability to add their own code into the ‘setup()’ and ‘main()’ segments, as required. The only stipulation here is that, to deal with external interrupts, user added code must utilise a specific ‘get interrupt’ process (i.e. ‘scan_IQ()’) to progressively action interrupts from the interrupt queue. This function will either return

1. no interrupts in the interrupt queue, or

2. the value of the generic interrupt handler that was triggered (0, ... ,5) (see below for an explanation of generic interrupt handlers). Additionally, a number of variables are set with all of the parameters defined in the configuration data for the respective external interrupt.

There are no other demands placed on the end user solution.

This interrupt framework is intended for use by anyone with an interest in ARDUINO development requiring external interrupts - novices and experts alike. The starting point for its use is to thoroughly understand what it is you want to do with external interrupts. That is, to understand:

a). why interrupts should be a part of your solution and, if so,
b). individually, how these should be configured for the solution

It is <u>essential</u> for a successful implementation of this framework that due regard and is time spent analysing the specific requirements for external interrupts and then to configure the framework suitably.

The remainder of this document provides further background and information about the framework, its origins, structure and use.

The Requirements (Objectives)

In designing this framework a number of objectives were set, these being to arrive at a solution for external interrupt handling such that:

- disassociates the event of an external interrupt occurring from the processing of its occurrence, providing freedom and control to the overlying end user application. That is, a solution that would provide independent processing from the ‘capture’ of received interrupts to their eventual processing
- code that could be ‘plugged’ into without extensive modification (a library would be ideal for this, but for now what is presented is an ARDUINO C++ program framework)
- is scalable, providing a framework to handle multiple external interrupts
- is highly configurable for each possible external interrupt. For example, the need to configure each interrupt differently if needed
- can support unique or nonunique external interrupt event capture/recoding prior to onward processing
- takes account of spurious interrupts. That is, provides debounce for ‘dirty’ switching.

A Solution (Method)

Generic Interrupt Handlers

By providing a framework that is independent of the specifics of a particular implementation of external interrupt handling it was necessary to introduce the concept of generic interrupt handlers. These generic interrupt handlers are ‘agnostic’ to any and all interrupts they are configured to service. The allocation of generic interrupt handlers is achieved by the configuration of a data structure that maps the ‘real-world’ to the virtual. Generic handlers are associated with individual external interrupts and processed according to their associated configured mapped parameters.

This mapping occurs through the table/array ‘interrupt_config_data’ such that each row of this array/table accords with a specific generic interrupt handler (0, ..., 5). For example, row 0 interrupt data is processed by generic_interrupt0(), row 1 interrupt data is processed by generic_interrupt1(), etc.

Interrupt Queue

The method employs the concept of a queue to ‘detach’ external interrupt events from their onward processing. The queue is managed on a first in, first out basis (FIFO) which ensures that external interrupts are dealt with in the order in which they occur (triggered).

To support this concept, a free chain of data blocks (each of two words) is created in accordance with the size configured (see ‘C00_Configurations Tab - Interrupt Queue Free Chain Size’, below).

When an external interrupt is triggered the following process occurs:

- If the external interrupt is defined as being ‘unique’ then an examination of the interrupt queue occurs to see if the interrupt already has an entry in the queue. If it does not then an entry of the event is inserted into the queue. If it does then it event is ignored
- If the external interrupt is defined as ‘nonunique’ then and entry is inserted into the queue

Free Chain

The Interrupt Queue (IQ) is ‘fed’ from a free chain of blocks that are allocated on demand by the generic interrupt handlers. The size of the free chain is determined by the configuration data value set in the configurations tab (see below C00_Configurations Tab - Interrupt Queue Free Chain Size’).

Configurable Parameters

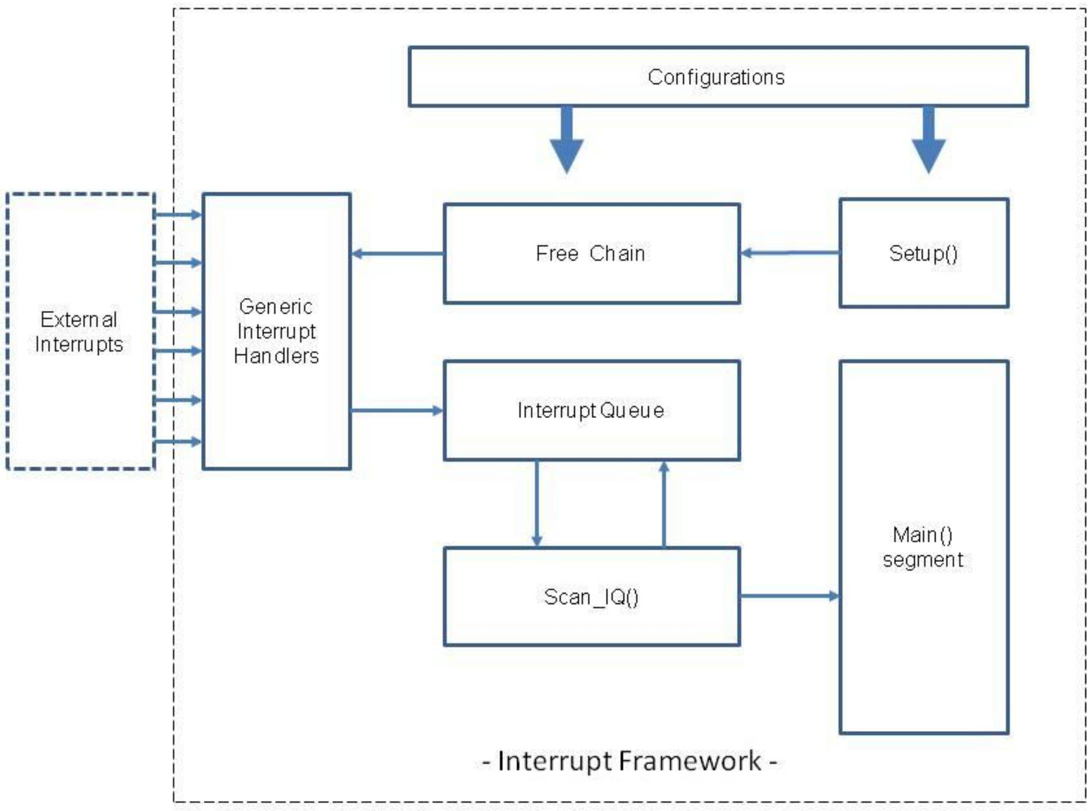
The framework is designed to be flexible and readily agreeable to configuration to meet end user external interrupt requirements (within the limits outlined). End user changeable parameters are maintained within one tab, this being C00_Configurations. See below for a full discussion of end users changeable parameters.

Framework Structure

The section presents an overview of the structure of the framework and how it can be used through configuration.

Diagramatic Conceptual Overview

At a conceptual level the interrupt framework can be viewed as below:



The framework is developed using ARDUINO IDE (version 1.8.12). The code is structured across a number of tabs to provide logical segmentation. These are:

1. A00_Interrupt_Framework_README_vx.xx
2. C00_Configurations
3. E00_Queue_Handlers
4. E10_Interrupt_Handlers
5. E90_Diags
6. G00_Setup
7. H00_Main_Segment
8. T00_Testing

End User Configurability

For end users, the only tab segments requiring any modification are:

- C00_Configurations
- E90_Diags
- G00_Setup

- H00_Main_Segment

The framework is designed such that end user solutions must ensure that the Interrupt Queue is regularly serviced. To achieve this, the framework provides a structure into which an end user code can be added.

End user code (the end user solution) should be inserted at tabs G00-Setup, E90_Diags and H00_Main_Segment. There is no further discussion regarding this, other than to say DO NOT remove or change any existing and resident variables or code in these segments. Simply insert end user code where indicated and/or appropriate.

Configuration of the framework occurs exclusively in the C00_Configurations tab. The remainder of this section therefore provides an overview of some of the possible ways in which external interrupts and the framework generally may be configured.

There are three possible end user changeable parameters that may be varied according to need. These are:

1. Diagnostics - a #define declaration that toggles if diagnostics reporting is on or off
2. Interrupt Configuration Data - an array/table that describes the specific configuration parameters for each possible external interrupt up to the maximum number of digital interrupt pins possible (for the ARDUINO MEG 2560 this is six)
3. Interrupt Queue Free Chain Size - a #define declaration that specifies the size of the free chain queue used to store interrupts as they are triggered

These are described below:

C00_Configurations Tab - *Diagnostics*

Parameter definitions:

Declaration	**Meaning**
`#define bool diags_on = true;`	the permissible values for this flag are 'true' or 'false'. If set to 'true' (diagnostics on) then the routines predefined in the E90_Diagnostics tab will be actioned if called. There are two routines available to the end user to aid diagnostics: 1. print_free_chain() – this routine prints the free chain of blocks plus its start of chain pointer. 2. print_IQ() – this routine prints the interrupt queue which shows all interrupts in the queue with associated interrupt values. Note that the above two routines will disable interrupts during their run. This same flag may be similarly used for any other diagnostic code required by the end user.

C00_Configurations Tab - *Interrupt Configuration Data*

Parameter definitions:

Declaration	**Meaning**
`volatile int interrupt_config_data [max_digital_inputs][7]`	This is the principal table (array) controlling how external interrupts are configured. Note that the number of rows parameter ('max_digital_inputs') is preset and should NOT

Declaration	**Meaning**
	be changed without some extension of existing code. This value is set to the maximum number of interrupts possible on the MEGA 2560 board (6). The rows of this table contain the necessary configuration data for each external interrupt to be defined. Note that the order of the row interrupt data in this array/table is not dependent of the priority of the external interrupts. External interrupt data may be entered in any order, row by row. This is because external interrupts are allocated a generic interrupt handler during setup(). Ordering external interrupts during configuration is therefore not important or a requirement of the framework.

'interrupt_config_data' column definitions:

Declaration Presets	**Meaning**
`Table (Array) Column Data`	The column data of each of row is interpreted identically as follows below:
`Column 0, active/inactive`	This value defines if this external interrupt (defined by the row) is active or not. If not used then this should be set to 'inactive'.
`Column 1, digital pin number`	This specifies the ARDUINO pin number associated with the external interrupt defined by this row. See ARDUINO data specification for what pins are mapped to what interrupts. For the MEGA 2560 these are detailed at append A.
`Column 2, pinMode mode value`	During setup(), each 'active' external interrupt is defined via a pinMode call. This call uses the pinMode mode value defined by this value. See Appendix B for mode types.
`Column 3, unique/nonunique flag`	If set to 'true' then the interrupt handlers will not insert an external interrupt triggered event into the interrupt queue if one already exists (unique). If set to 'false' then multiple external interrupt trigger events may exist in the interrupt queue (nonunique). Clearly, each interrupt can be set to either value allowing a mix on unique and nonunique processing if required.
`Column 4, trigger type`	External interrupts are linked to digital pins and ISRs during the setup() process. At this time, the attach interrupt call will use the value held in this column to determine the type of trigger required. See Appendix C for trigger types.
`Column 5, debounce value`	When external interrupts trigger it is often the case that spurious interrupts are also raised by the physical event that led to the interrupt. This parameter ('debounce') provides a window during which external interrupts arriving are ignored. The value defined here is in milliseconds (msecs).
`Column 6, external interrupt`	This defines the external interrupt number that is linked to defined digital interrupt pin.

Example External Interrupt Parameters

```
volatile int interrupt_config_data[max_digital_inputs[1]][7]=
{
// 0         1          2             3        4      5    6   comment
active,     21,   INPUT_PULLUP,   RISING,   true,   15,   0   // generic interrupt
                                                               handler 0 entries
active,     20,   INPUT_PULLUP,   RISING,   false,   5,   1   // generic interrupt
                                                               handler 1 entries
active,     19,   INPUT_PULLUP,   RISING,   true,   15,   2   // generic interrupt
                                                               handler 2 entries
inactive,   18,   INPUT_PULLUP,   RISING,   true,   15,   3   // generic interrupt
                                                               handler 3 entries
active,      2,   INPUT_PULLUP,   RISING,   false,   5,   4   // generic interrupt
                                                               handler 4 entries
active,      3,   INPUT_PULLUP,   RISING,   true,   15,   5   // generic interrupt
                                                               handler 5 entries
}
```

C00_Configurations Tab - *Interrupt Queue Free Chain Size*

A free chain of data blocks (2 words each) is created during the set up process, the number of blocks being defined by the variable 'max_IQ_free_chain_blocks'. Initially, the free chain is configured in accordance with the value defined by the declaration:

Parameter definitions:

Declaration	**Meaning**
`#define max_IQ_free_chain_blocks 64`	The free chain will be created with this number of entries (blocks). Each free chain block comprises 2 words (integers). The first word of each free block contains a forward pointer to the next free block or -1 (`end_of_chain_value`) if the last free block. In this example, the free chain would be created as 64 available blocks to be allocated to the Interrupt Queue (IQ) on demand. However, it can also be seen that some interrupts in the above example are defined to be nonunique. That is, more than one interrupt event can be in the IQ at the same time. If nonunique interrupts are required then the value of this #definition must be carefully considered, otherwise, triggered interrupts may not be able to be inserted into the interrupt queue if it is too small. Selecting a value may not be straight forward and trial and error may be needed dependent on the specifics of the application being developed. However, unless 100's of interrupts are being generate per second then you may find that a value of, say, 16, 32, 64 or 128 would be of adequate size for the free chain. Note that it is not a requirement that free chain size is a power of 2. It can be any value, e.g. 17, 29, etc.

[1] max_digital_inputs – this is set to 6 for this framework, reflecting the number of external interrupts supported by the MEGA 2560

C00_Configurations Tab - *max_digital_inputs*

There is an additional parameter defined within the C00_Configurations tab, but this is not for end user modification. It is mentioned here for completeness, only.

Parameter definitions:

Declaration	**Meaning**
`#define max_digital_inputs 6`	This declaration is included here for completeness – it is not an end user changeable parameter. This declaration defines the number of external interrupts that the framework is designed for - this reflects the number of generic interrupt handlers defined. For the current version of the framework this is the maximum number of external interrupts available on the ARDUINO MEGA 2560 microcontroller, hence it is six. See Appendix A for a list of the MEGA 2560 digital pins that support external interrupts.

Performance of Interrupt Queue Handling

It goes without saying that dealing with interrupts via way of an intermediary queuing system must come with some overhead baggage, this being in storage space and execution time. Where an application is time critical in dealing with external interrupts then it may be better to design the solution directly such that ISRs form the core of the construction.

However, to aide this judgement, some performance measurements have been taken to understand the timings for interrupt queue management. Testing examined:

a. inserting interrupts into the interrupt queue until it was fully saturated (fully loaded),
b. removing interrupts successively from the interrupt queue until it was fully emptied.

Tests were run for different sizes of interrupt queues – 1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024[2] queue entries. What was found was that timings were largely not at all affected by changing the size of the free chain/interrupt queue. To ensure optimal processing, the interrupt debounce value was set to 0 milliseconds, which may not be real world justified for some uses, and with unique interrupt processing turned off (allowing any number of the same interrupts in the interrupt queue at the same time).

The results obtained showed that:

1. the average time to insert an interrupt into the interrupt queue is 15 microseconds,
2. the average time to remove an interrupt from the interrupt queue is 12 microseconds.

Clearly, a little less work taking out from the queues than putting in!

Global Data and Code Sizes

Measurement were taken to determine the respective sizes of global data and framework code, with and without diagnostic code included.

The only variable data impacting on global data allocation relates to the size of the free chain initialised for the interrupt queue. The base value taken for determining sizing was 16 free chain blocks (i.e. 64 bytes of data requirement). This provided the following results:

[2] There is nothing special about this size progression. The queue size can be any value.

Sizing Metrics With Diagnostic Code

Total allocation size for global data including the free chain:	409 bytes	(this value includes the free chain sized at 16 blocks, each of 4 bytes (total 64 bytes).)
Total size of generated code, excluding global data:	5,392 bytes	
Total size (data and code)	5,801 bytes	

Sizing Metrics Without Diagnostic Code

Total allocation size for global data including the free chain:	234 bytes	(this value includes the free chain sized at 16 blocks, each of 4 bytes (total 64 bytes).)
Total size of generated code, excluding global data:	4,346 bytes	
Total size (data and code)	4,580 bytes	

Free Chain Sizing Influences

Clearly, the greater the size of the free chain the more global memory will be allocated. However, the relationship between the number of free chain blocks and additional memory requirements is completely linear. That is, using sizing data provided above, then if a free chain size of 32 blocks is required then the respective additional global data size increase by 4 x 16, i.e. 64 bytes, as the sizing above already includes 16 free chain blocks. The logic is similarly applied for any other free chain size. It is simply:

Additional global data space = (Required size (in blocks) – 16[3]*) x 4 bytes*

For each additional free chain block increase above the initial 16 configured for the sizing model the global data requirement will increase by 4 bytes.

Using these metrics it will therefore be possible to determine how global space will be allocated just for the framework alone, this being a function purely of the free chain size.

Guidance for Implementation

This section provides guidance to assist with implementing this framework.

General Considerations

Applying this framework to the development of an application requires a number of general considerations to be made:

1. Configuration of framework parameters and settings described above at End User Configurability - Configurable Parameters. This should be done following a thorough analysis of requirements for external interrupts, for example
   a. how many external interrupts are needed ?
   b. can any of the interrupts be defined as 'unique' in the IQ until processed?
2. What are the likely interrupt frequencies as this may impact on the size of the free chain?
3. What code is needed to be included in the set_up() segment?
4. What code is needed to be included in the main segment, code that will handle:

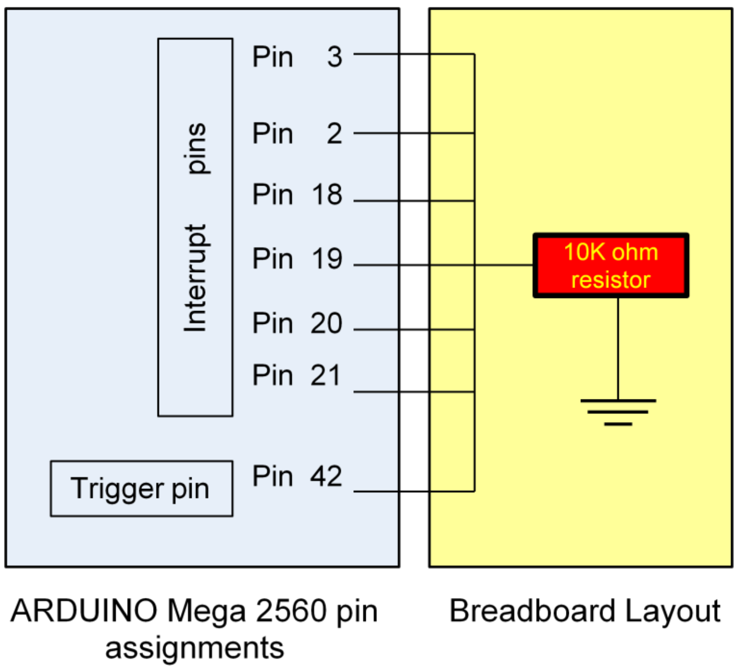
[3] 16 free chain blocks already included in based sizing model

   a. external interrupt processing?
   b. any processing needs that are relevant when no interrupt alerts are current?
5. Note that there are very few end user functions to interface with; there is only one principal one and two ancillary ones. See Appendix E, End User Framework Functions.
6. What additional unique to application requirements are necessary for functions? Where this is so, then add additional tabs into the programme such that the tab name is inserted prior to the set_up and main segments so that these will be in scope when used in the set_up() and main segments.
7. When an interrupt alert is obtained from a scan_IQ() call, then the end user will have the following global variables available for use:

`int_number`	- the actual real world interrupt number
`int_pin`	- the digital pin number triggering the interrupt
`int_pinmode`	- the pinMode value used to set up this digital interrupt pin
`int_trigger`	- the interrupt trigger value used to set up this digital interrupt
`int_unique`	- the flag value defining if interrupt processing is unique or nonunique
`int_debounce`	- the debounce value in msecs for this interrupt

Example – Multiple and Simultaneous External Interrupts

To help apply the concept promoted by the framework and to assist in understanding how it can be implemented, an example is provided at Appendix F (Interrupt Simulation).
In this example, all six possible external interrupts for the Mega 2560 microcontroller are configured and wired to a simple breadboard circuit design that will simulate triggering events under program control simultaneously. The breadboard circuit design is:



To note is that digital pin 42 (no reason why it is this pin) is toggled HIGH and then LOW. This simulates a rising trigger event and this is presented to each interrupt pin simultaneously. That is, all interrupt pins are raised and lowered at the same time.

What happens is that the microcontroller will deal with each interrupt event in its order of priority defined by the microcontroller specification. That is, interrupt 0 (pin 21) will be processed first, then interrupt 1 (pin 20) next, and so on, until the last one, interrupt 5 (pin 3).
The output to the serial monitor will show the sequence of processing by the framework as interrupts 0, 1, 2, 3, 4 and 5 (actual interrupt numbers) which is as would be expected.

The example is driven by the interrupt configuration data set in the array/table '`interrupt_config_data`' (see Appendix F).

Each interrupt is configured as being nonunique, INPUT_PULLUP, RISING and with a debounce setting of 0 (we want maximum throughput).

It should be appreciated that the order of the entries in this array/table can be random. They do not need to be arranged in any order of priority. This is because each external interrupt is assigned to an internal generic interrupt handler which handles it purely on the configured data for its associated interrupt. As an exercise, the reader may wish to rearrange the entries and witness that the serial monitor will confirm that they are dealt with in the correct order (priority).

Finally, the design of the simulation means that there will be no more than six interrupts unprocessed in the interrupt queue at any one time. Therefore, the size of the free chain can be safely set to six entries.

Conclusion

The development of the interrupt framework was steered by the objectives established at the outset. The final product has met these objectives as follows:

Objective	**How Met**
disassociates the event of an interrupt occurring from the processing of its occurrence, providing freedom and control to the overlying end user application. That is, a solution that would provide independent processing from the 'capture' of received interrupts to their eventual processing.	Object met. The solution is based on the concept of a queue of interrupts that can be processed after an interrupt event and at a time dictated by the end user application.
code that could be 'plugged' into without extensive modification.	Objective met. End user coding occurs in the setup() and main() segments. Additional segments may be added/included as necessary without change to any framework element.
is scalable, providing a framework to handle multiple external interrupts	Objective met. The framework will handle from one to six concurrent interrupts, this being constrained by a design based on the MEGA 2560 microcontroller. Further interrupt handling can be provided by suitable extension. The solution is therefore extensible.
is highly configurable for each possible interrupt. For example, the need to configure each interrupt differently if needed	Objective met. Interrupt handling is highly configurable depending on end user requirements. Each of the six external interrupt definitions can be uniquely configured in required.

Objective	**How Met**
can support unique or nonunique interrupt event capture/recoding prior to onward processing	Objective met. Interrupt handling is designed to allow external interrupt events to be recorded as 'unique' or 'nonunique' and suitably recorded within the interrupt queue.
takes account of spurious interrupts. That is, provides debounce for 'dirty' switching.	Objective met. Generic interrupt handlers are coded to take account of 'interrupt bounce' as may occur, for example, from a 'dirty' switch. Debounce can be configured for each external interrupt in milliseconds (msecs). If set to 0 msecs, then all interrupt events will be record, unless configured as 'unique'. Note also that the debounce code takes regard of the millis() function wrap around when it overflows back to 0 msecs.

Whether the framework proves to be helpful in implementation will depend on the specific needs of the end user solution sought for the capture and processing of external interrupts. Although the performance data gathered (see above) suggests that the method is, on the whole, efficient and provides good levels of performance, it use must be judged as to whether it is applicable for the need.

If absolute performance is the principal requirement then perhaps a directly crafted solution is appropriate, one with no/few overheads. However, it is considered that for most every day requirements the framework will provide a very helpful basis on which to deal with external interrupts in an application.

APPENDICES

Appendix A

ARDUINO MEGA 2560 External Interrupt to Pin Assignments, by Priority

External Interrupt Number (by priority)	Pin Assignment
INT0	21
INT1	20
INT2	19
INT3	18
INT4	2
INT5	3

Appendix B

ARDUINO pinMode Mode Values

Mode Values
INPUT
INPUT_PULLUP
OUTPUT

Appendix C

ARDUINO Interrupt Trigger Values

Trigger Values
LOW
CHANGE
FALLING
RISING
HIGH[4]

Appendix D

ARDUINO Microcontroller Boards with Interrupt Pin Assignments

Microcontroller Board	Digital Pins Usable for Interrupts
Uno, Nano, Mini, other 328-based	2, 3
Uno WiFi Rev.2	all digital pins
Mega, Mega2560, MegaADK	2, 3, 18, 19, 20, 21
Micro, Leonardo, other 32u4-based	0, 1, 2, 3, 7
Zero	all digital pins, except 4
MKR Family boards	0, 1, 4, 5, 6, 7, 8, 9, A1, A2
Due	all digital pins
101	all digital pins (only pins 2, 5, 7, 8, 10, 11, 12, 13 work with CHANGE)

[4] Due, Zero and MKR1000 boards only

Appendix E

End User Framework Functions

The framework provides a number of essential end user accessible functions to develop applications around the use of external interrupts. These are described below:

print_free_chain()

Purpose:	To print the entire free chain of blocks, helpful for diagnostics when an application is under development. Note that the Boolean flag 'diags_on' must be set to true for the function to print.
Parameters:	none
Return value:	none

Example
```
print_free_chain();
```

print_IQ()

Purpose:	To print the entire chain of allocated free blocks in the interrupt queue. Helpful for diagnostics when an application is under development. Note that the Boolean flag 'diags_on' must be set to true for the function to print.
Parameters:	none
Return value:	none

Example
```
print_IQ();
```

scan_IQ

Purpose:	Examines the end of the interrupt queue (FIFO) and if there is an interrupt in the queue waiting, will remove it and set up end user parameters: int_number - the actual real world interrupt number int_pin - the digital pin number triggering the interrupt int_pinmode - the pinMode value used to set up this digital interrupt pin int_trigger - the interrupt trigger value used to set up this digital interrupt int_unique - the flag value defining if interrupt processing is unique or nonunique int_debounce - the debounce value in msecs for this interrupt
Parameters:	none
Return value:	Either: the <u>generic</u> interrupt number (>=0), or no_interrupt_request (-1)

Example
The main segment code is as follows (blue text is end user code added to the base framework):

```
void loop()
{int  interrupt;
/*  The following section of code will allow interrupts to be processed at
each loop cycle.
Add whatever other code for the specific application required to process the
interrupts received plus any other requirements of a noninterrupt nature.

Design your code structure as necessary, but note that to obtain interrupts
from the interrupt queue it is necessary to call the routine 'scanIQ()'.
This will examine the interrupt queue and if an unprocessed interrupts exists
the result of the call returns the generic interrupt number and associated
interrupt data that triggered the interrupt, if available.

For example, 'interrupt=scanIQ();'.
Results of this call will be:

1.  an interrupt has been obtained from the IQ.  In which case the following
    variables will be set up:

    int_number   - actual real world interrupt number
    int_pin      - actual digital pin number triggering the interrupt
    int_pinmode  - pinMode value used to set up this digital interrupt pin
    int_trigger  - interrupt trigger value used to set up this digital
                   interrupt
    int_unique   - flag that defines if interrupt processing is unique or
                   nonunique
    int_debounce - debounce value in msecs for this interrupt

2. if no interrupt is returned (none in the IQ) then all of the above
   variables will be set to 'no_interrupt_request'.
*/
do { // Keep processing interrupts whilst there are interrupts in the queue..
   interrupt = scan_IQ();  //  get the next interrupt in IQ if there is one.
   if (interrupt != no_interrupt_request)
   {/*
    Process this interrupt request. 'interrupt' defines the
    generic interrupt number that triggered.  Other variables
    (see above) give all other interrupt attributes if required
    Insert whatever code appropriate here, if any,
    when not processing an interrupt request  */
    switch (int_number)
    {
      case 0: // external interrupt 0
        // place your code for this interrupt number here...

        break;
      case 1: // external interrupt 1
        // place your code for this interrupt number here...

        break;
      case 2: // external interrupt 2
        // place your code for this interrupt number here...
        break;
      case 3: // external interrupt 3
        // place your code for this interrupt number here...

        break;
      case 4: // external interrupt 4
        // place your code for this interrupt number here...

        break;
      case 5: // external interrupt 5
        // place your code for this interrupt number here...

        break;
      default:
        break;
    }
   }
  }
  while (interrupt != no_interrupt_request);
  /*
  No interrupts left in the queue, so do other things....
  Insert whatever code appropriate here, if any,
  when not processing an interrupt request. */

}
```

Appendix F

Example – Interrupt Simulation

The following provides a simple example of the way in which the framework operates.
It is designed so that all available interrupts (six on the Mega) are triggered simultaneously and dealt with individually. In this case, each interrupt will be processed in its order of interrupt priority defined by the microcontroller specifications.

The configuration data for the simulation is as follows (C00_Configurations tab):

```
volatile int interrupt_config_data[max_digital_inputs][7]=
{
active, 21, INPUT_PULLUP, RISING, false, 0, 0,
active, 20, INPUT_PULLUP, RISING, false, 0, 1,
active, 19, INPUT_PULLUP, RISING, false, 0, 2,
active, 18, INPUT_PULLUP, RISING, false, 0, 3,
active,  2, INPUT_PULLUP, RISING, false, 0, 4,
active,  3, INPUT_PULLUP, RISING, false, 0, 5};

//  ****Interrupt Queue configurable values ****
#define max_IQ_free_chain_blocks 6
```

The set_up() tab code is as follows (blue text is end user code added to the base framework):

```
void setup() {
  int input, interrupt_pin, pinmode_type, trigger_type;
  initialisation_complete = false;  //  ensures no interrupts are processed
                                        before ready
  //  set up the free chain for interrupt queue handling
  start_of_IQ_chain       =  -1;    //  start with no interrupt blocks in IQ
                                        chain
  end_of_IQ_chain         =  -1;    //  start with no interrupt blocks in IQ
                                        chain
  num_IQ_interrupts       =   0;    //  start with no interrupt blocks in IQ
                                        chain
  create_IQ_free_chain();           //  set up the free chain of IQ blocks
  if (diags_on)
  {
    Serial.begin(115200);
  }
  //  establish the interrupt handler routines to the defined input pins
  //  and associated parameter values.

  for (input=0; input < max_digital_inputs; input++)
  {
    if (interrupt_config_data[input][0] == active)
    { // This interrupt and associated pin is active, so set up structures for it
      interrupt_pin     =interrupt_config_data[input][1];
      pinmode_type      =interrupt_config_data[input][2];
      trigger_type      =interrupt_config_data[input][3];
      pinMode(interrupt_pin,pinmode_type);
      attachInterrupt(digitalPinToInterrupt(interrupt_pin),
                      interrupt_handler_addresses[input],
                      trigger_type);
    }
  }
  //
  //  Place any other set up code here, but BEFORE 'initialisation_complete =
      true'
  //

  //  set up digital pins to artificially toggle high then low
  //  this will cause an interrupt to be raised on the configured interrupt pin
  //  defined in the configuration table/array
  //
  //  note, there is no reason why pin 42 output pin was chosen,
  //  choose any pin that is convenient.
  pinMode(42,OUTPUT);
  digitalWrite(42,LOW); // force low state

  //
  //  End of user inserted set up code
  //
  initialisation_complete = true;   // can now allow interrupts to be processed
}
```

The main segment code is as follows (blue text is end user code added to the base framework):

```
void loop()
{int  interrupt;
/*  The following section of code will allow interrupts to be processed at each
    loop cycle.  Add whatever other code for the specific application required to
    process the interrupts received plus any other requirements of a noninterrupt
    nature.
    Design your code structure as necessary, but note that to obtain interrupts
    from the interrupt queue it is necessary to call the routine 'scanIQ()'.
    This will examine the interrupt queue and if an unprocessed interrupts exists
    the result of the call returns the generic interrupt number and associated
    interrupt data for the triggered interrupt.
    For example, 'interrupt=scanIQ();'
    Results of this call will be:
    1.  an interrupt has been obtained from the IQ.  In which case the following
        variables will be set up:
          int_number    - actual real world interrupt number
          int_pin       - actual digital pin number triggering the interrupt
          int_pinmode   - pinMode value used to set up this digital pin
          int_trigger   - interrupt trigger type used to set up the digital pin
          int_unique    - defines if interrupt processing is unique or nonunique
          int_debounce  - debounce value in msecs for this interrupt

    2.  no interrupt is returned (none in the IQ) then all of the above variables
        will be set to 'no_interrupt_request'.
*/
do { // Keep processing interrupts whilst there are interrupts in the queue..
    interrupt = scan_IQ();  //  get the next interrupt in IQ if there is one.
     if (interrupt != no_interrupt_request)
     { /*  Process this interrupt request. 'interrupt' defines the
           generic interrupt number that triggered.  Other variables (see above)
           give all other interrupt attributes if required.
           Insert whatever code appropriate here, if any,
           when not processing an interrupt request  */
         Serial.print("int fired..int no: ");
         Serial.println(int_number);
         Serial.flush();
       }
     }
  while (interrupt != no_interrupt_request);
  /*  No interrupts left in the queue, so do other things....
      Insert whatever code appropriate here, if any,
      when not processing an interrupt request. */
  // Artificially generate all interrupts, toggle nominated pin high then low.
  // The circuit design will cause all interrupt defined pins to go HIGH
  // and then LOW, causing a trigger event.
  digitalWrite(42,HIGH);
  digitalWrite(42,LOW);
  //delay(50);    // enable and vary this if required, but works okay without it
}
```