Arduino

ez Serial/Parallel IC Library (SIPO8) User Guide

A Virtual Library Supporting the
Configuration & Driving of
Multiple Serial/Parallel ICs
(SIPO ICs)

Author: R D Bentley, Stafford, UK.
Date: April 2021
Version: 1.00

Warranties & Exceptions

Change Record

Version	Date	Change
1.00	April 2021	Initial version published

Audience

To fully understand and make good use of this User Guide and the capabilities of the <ez_SIPO8_lib> library, the reader should already be familiar with 8bit serial/parallel ICs, such as the 74HC595[1] IC. Familiarity with wiring these ICs and driving them with suitable Arduino code, for example the standard `shiftOut` function, provides an excellent basis on which to build and develop sophisticated and innovative solutions through the use of the <ez_SIPO8_lib> library.

Crib Sheet

A highly reduced version of the essential parts of this User Guide can be downloaded in crib sheet form from github.

1 Throughout this User Guide links to internet resources will be provided where further information may be considered to be helpful. In so doing, there is no intention to promote any supplier or product.

Contents

Introduction

This User Guide (UG) provides guidance and key technical information to assist in developing Arduino projects incorporating Serial In-Parallel Out ICs (SIPO ICs[2]); for example, the ubiquitous and low cost 74HC595 IC, an 8bit device, or any other SIPO IC that operates with the same 3-wire interface (see SIPO8 Library Design Architecture, below).

The library operates in a virtual manner, insofar as physical SIPO output pins are mapped to software structures and updated independently using a wealth of functions/methods, until a physical update is desired/required.

The library provides many functions/methods to manipulate output pins using absolute and relative pin addressing. Physical SIPO ICs are grouped into banks of output pins, single SIPO ICs or in cascaded form. There is no practical limit to the number of SIPO ICs that may be cascaded in a single daisy chain, even beyond eight.

Essential to use of the library and successful implementation will be an understanding of the SIPO8 library design architecture (see below), which provides a powerful an innovative solution to the driving of very many outputs using few microcontroller digital pins.

This UG limits itself to the use and implementation of the SIPO8 library capabilities, it does not cover:

- considerations regarding the connection of SIPO outputs to components other than standard LEDs, e.g. relays or other components – *care needs to be applied here in matching the particular SIPOs used to the components they are to connect to, together with power sources*.

The internet provides a significant amount of material covering the above and the reader is recommended to do a little research around such issues before undertaking anything but simple projects using LEDs.

However, these matters aside, this UG should make life a good deal easier in designing projects incorporating SIPO ICs, singly or in multiple cascaded groupings. Indeed, to supplement understanding and end user design innovation, a number of tutorials are also provided, each dealing with specific and essential aspects of the design concepts and the library's rich set of capabilities/methods.

[2] The term SIPO will be used throughout this UG to refer either a physical IC or virtual software control structures designed to map the physical characteristics of configured SIPO ICs. The context within which the term is used will make it clear which is referenced.

Overview

Design Objectives

At the outset, a number of key objectives were established for the design of the SIPO8 library, these being a library that provided/supported:

- a design that would maximise the number of physical output pins that could be supported/driven by the standard range of Arduino microcontrollers
- a representation of physical SIPO ICs and their output pins into the virtual, allowing independent programmatic manipulation of outputs
- a logical, flexible and straight forward design - one that would allow the end user developer to easily manage/control a large number of outputs discretely, or as a set of outputs; a design architecture that would be modular and granular
- recognition that many common SIPO ICs have a standard hardware interface which could be leveraged and used to good advantage – SIPO IC hardware transparency
- a solution that would be scalable, allowing freedom for the end user to incorporate many SIPO ICs in their project design
- a solution that provides support for many 100s of outputs
- a rich set of functional capabilities to define, control and drive SIPO ICs, individually or in a cascaded arrangement, as individual banks
- the capability to group, through cascading/daisy chaining, any number of SIPO ICs in a single arrangement (bank), even beyond eight ICs in a single cascade
- the capability to define any number of differently sized SIPO banks, single or multiple and cascaded SIPO ICs
- the capability to work with SIPOs of different and mixed bit/pin size, e.g. 8, 16, 32 or 64bit SIPO ICs
- tools for the end user to examine key data during development

The next section provides an overview of the library's design architecture. It is this that provides and meets all of the above design objectives.

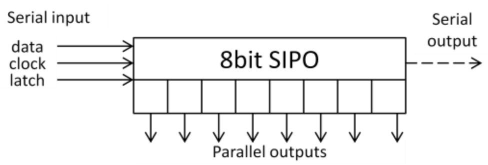
SIPO8 Library Design Architecture

Basic Design

The architecture design is based on a generic set of capabilities of a standard 8bit serial/parallel IC (SIPO IC), such as the 74HC595 IC (others are available). In particular, the following SIPO IC characteristics are at the core of the architecture:

- Serial interface – 3-wire digital interface for driving data, clock and latch signalling
- 8bit parallel outputs
- Cascade features (linking SIPO ICs together in a daisy chain arrangement to drive more outputs on a single 3-wire serial interface)

Diagrammatically, these characteristics are:



Any SIPO IC with the above characteristics should be suitable for use with this library, even 16, 32 or 64bit SIPO ICs, as these may be simply broken down into basic 8bit SIPO building blocks and configured into banks (see below). That is, you may construct a bank of mixed bit size SIPOs, if required, e.g. 1 x 8bit + 1 x 32bit = 2 SIPOs/40 bits, which will equate to a bank size of 40/8 = 5 standard 8bit SIPOs.

However, the choice of SIPO IC type must be determined by what will be connected to the outputs; *it is essential to match the power requirements of both the connected components and SIPOs*. These matters are out of scope for this UG, but suitable guidance can be found on the internet.

Although the architecture design is based on a concept of a generic 8bit SIPO IC, this does not mean that it does not provide sophistication and flexibility in supporting many SIPO ICs with many 100s of outputs, or SIPO ICs of other bit sizes. Indeed, at its design limit, the library will support up to 255 8bit SIPO ICs, a total of 2040 output pins, or a combination of other bit sized SIPOs totalling and equivalent to 255 8bit SIPO ICs.

To accomplish such flexibility and scale, the architecture is designed around a hierarchical construct as follows.

- all SIPO output pins collectively form an array.
- the array is subdivided into banks of SIPO ICs.
- a bank of SIPO ICs may contain a single SIPO IC (8bit) or any number cascaded together.

The architecture design therefore looks like:

<table>
<tr><td>Level: array</td><td colspan="12">array of outputs</td></tr>
<tr><td>Level: bank</td><td colspan="9">bank 0</td><td>bank 1</td><td>...</td><td>bank n</td></tr>
<tr><td>Level: output pins</td><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td colspan="2">...</td><td>N</td></tr>
</table>

The array is the complete set of connected output pins, configured into one or more banks, with each bank comprising one or more SIPO ICs.

Addressability

During the library's class initiation, the class constructor function will create a virtual array map, or pool, for all output pins, based on the total number of SIPO ICs specified in the initiation process. For example, if the number of SIPO ICs specified is 10, then the bit map array would comprise 80 output pins with absolute addresses from output pin 0 to 79, inclusive. Output pin addresses are always referenced from 0.

However, it is not until the array pool is configured into SIPO banks, and therefore made active, that any of these array output pins will be addressable. Allocating SIPO ICs into banks brings into scope their respective output pins, thereby making them visible (active) and addressable. An efficient implementation will ensure that all array pool outputs are associated to a bank and therefore become active/addressable.

The library provides functions that allow us to address virtual SIPO output pins in two ways:

Absolute Addressability
Absolute addressability occurs at array level only – output pin addresses will range from 0 to the (total number of active output pins) – 1, inclusive. Functions using absolute addressing are: `set_array_pin`, `read_array_pin`, `invert_array_pin`.

Relative Addressability
Relative addressability occurs at bank level only – the output pins associated with a bank will range from 0 to the (total number of outputs pins mapped by a bank) – 1, inclusive.

To illustrate this, lets us consider the follow example:

We have created an array pool of 48 output pins (six SIPO ICs) and assigned these to three banks as follows:

bank 0, 1 x SIPO IC, 8 output pins,
- array pool pin addresses: 0 – 7 absolute address range
- bank 0 pin address range: 0 – 7 relative address range

bank 1, 2 x SIPO ICs, 16 output pins,
- array pool pin addresses: 8 – 23 absolute address range
- bank 1 pin address range: 0 - 15 relative address range

bank 2, 3 x SIPO ICs, 24 output pins,
- array pool pin addresses: 23 – 47 absolute address range
- bank 2 pin address range: 0 – 23 relative address range

... all array pins are assigned uniquely to a bank

Note the difference in addressing ranges – absolute addressing of the array is continuous from pin 0 to the very last pin, whilst relative addressing always starts at bank pin address 0 and continues incrementally to the last pin of the last SIPO IC defined by a bank. Banks therefore 'chop up' the array pool into well defined and unique subsets of SIPO output pins.

The library provides many functions to address any active output pin either at an array pool level or specifically within a bank. The are many functions using relative addressing, for example: `set_bank_pin`, `read_bank_pin`, `invert_bank_pin`, ..., etc.

Finally, and very important to fully appreciate, virtual SIPO output pins may be manipulated (set, inverted, read, tested, etc) using a wide range of functions/methods in isolation without changes being instantly reflected to their physical output pin equivalents. Physical SIPO output pins are only updated to reflect their virtual counterparts when required by using one or more xfer function calls. In this way, the virtual array pool of output pins may be manipulated/processed independently until such time/point that a physical update is required.

Once you get a handle of the design you will see its power, capabilities and flexibility.

Constraints & Limitations

Nothing in this world is perfect and <ez_SIPO8_lib> is not. Whilst it does provide a set of powerful and useful capabilities to aid and assist Arduino project developers involving SIPO ICs, there are several constraints and limitations in its design and use to be aware of:

1. the design is predicated on a standard and commonly available 8bit serial-parallel IC, such as the [74HC595](74HC595) IC. The smallest word size supported is therefore 8bits, albeit the design allows much larger outputs to be grouped through 'banking' into word lengths as a multiple of 8bits. For example, banks of size 8, 16, 32, 40, 48, 56, 64bits, etc.
2. The microcontroller/SIPO IC interface is driven by three digital output pins – data, clock and latch signals, controlling the transfer of data between the microcontroller and SIPO ICs
3. A maximum of 255 8bit SIPO ICs may be configured, giving a theoretical maximum of 2040 output pins which may be physically arranged into independently size banks. Testing with a Mega 256 suggests that the SIPO8 class may be configured to this maximum value, <u>including</u> the maximum number of timers (255). Lesser microcontrollers will dictate lower maxim ceilings.

Using the <ez-SIPO8_lib> Library

The SIPO8 library is used in a straight forward manner, the essential requirement is to understand its design architecture, including the concept of banking and the differences between absolute and relative addressing. The following sections provide guideance in applying the SIPO8 library to your projects

Location of the <ez-SIPO8_lib> Library

The SIPO8 library files should be installed within a directory called 'ez_SIPO8_lib' under the Arduino libraries directory - ...\Arduino\libraries\.

The <ez_SIPO8_lib> directory may comprise five files, three mandatory and two optional:

<u>Mandatory files</u>:
1. ez_SIPO8_lib.h ... header file
2. ez_SIPO8_lib.cpp ... C++ functions
3. keywords.txt ... keyword file to highlight <ez_SIPO8_lib> assets

<u>Optional files</u>:
4. ez_ SIPO8_lib_user_guide.pdf ... this document, or file elsewhere if required,
5. ez_ SIPO8_lib crib sheet.pdf ... an easy reference for key library data/functions,

Steps to Successful Use

Before 'flighting to task', it is recommended to think carefully about what it is you wish to achieve, how SIPO ICs are incorporated into your project and how <ez_SPIB8_lib> can be utilised.

The principal considerations are:

1. decide how many SIPO ICs and of what type?
2. how SIPO ICs will be arranged into banks, and for each bank, which microcontroller digital pins will be used for each bank (3 digital pins per bank)?
3. what will each physical SIPO output pin be connected to, e.g. LED, relay, etc. *Do your research if connecting these SIPO output pins to anything other than a LED*
4. do the physical SIPO ICs need to be powered from an external source? *Again, do your research*

If you are implementing more than one or two SIPO ICs it may be helpful to make a note of their configurations, as once you start wiring and coding things can get a bit muddled up! The following template may be helpful to fill out at the start of your planning and for you to refer to into the development stage (it is also a useful documentation aid post implementation):

<table>
<tr><td>Project:</td><td colspan="6"></td><td>Date:</td><td></td></tr>
<tr><td colspan="4">User Defined Parameters</td><td colspan="3">create_bank Results</td><td colspan="2" rowspan="3">Comments</td></tr>
<tr><td colspan="3">Control Pins</td><td rowspan="2">Num SIPO ICs in Bank</td><td rowspan="2">Bank Num</td><td rowspan="2">First Pin</td><td rowspan="2">Last Pin</td></tr>
<tr><td>Data</td><td>Clock</td><td>Latch</td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2"></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2"></td></tr>
<tr><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td colspan="2"></td></tr>
</table>

(add more rows as needed)

Start by filling out the white sections and use the **print_SIPO_data** function after you have created the banks to complete the central panel (**create_bank** Results). You will now have a written record of your SIPO configuration.

Example:

<table>
<tr><td>Project:</td><td colspan="6">LED indicators, with temperature output to pair of single matrix displays</td><td>Date:</td><td>24/04/2021</td></tr>
<tr><td colspan="4">User Defined Parameters</td><td colspan="3">create_bank Results</td><td colspan="2" rowspan="3">Comments</td></tr>
<tr><td colspan="3">Control Pins</td><td rowspan="2">Num SIPO ICs in Bank</td><td rowspan="2">Bank Num</td><td rowspan="2">First Pin</td><td rowspan="2">Last Pin</td></tr>
<tr><td>Data</td><td>Clock</td><td>Latch</td></tr>
<tr><td>2</td><td>3</td><td>4</td><td>1</td><td>0</td><td>0</td><td>7</td><td colspan="2">Console bank, sensor output LEDs, each LED tracking different input sensors for low/high watermark readings. Sensor inputs on analogue pins A0-A5.</td></tr>
<tr><td>5</td><td>6</td><td>7</td><td>2</td><td>1</td><td>8</td><td>23</td><td colspan="2">7 segment LEDs, 2 x banks - 10's & units for temperature respectively</td></tr>
</table>

Using the `<ez_SIPO8_lib>` is no different to using any other Arduino library. There are a few things we need to adhere to:

1. we need to ensure our sketch references the library
2. we need to create an instance of the library class, and
3. we need to understand how to correctly use the library's capabilities (e.g. functions and data).

... as follows:

Step 1

To start, we need to declare the `<ez_SIPO8_lib>` library. At the top level of your sketch include the following statements:

```
#include <Arduino.h>
#include <ez_SIPO8_lib.h>
```

Step 2

Then, prior to `setup()`, we need to declare and initiate the **SIPO8** class. This will be something like:

```
SIPO8 my_SIPOs(number_SIPOs, number_timers);
```

... where

`my_SIPOs` is the name you wish to give to the class and which will be used whenever you wish to access/use any of the SIPO8 resources (see later).

`number_SIPOs` is the number of 8bit hardware SIPO ICs you wish to size the class instance for. Note that this defines the maximum size of the virtual array output pin pool. It is the maximum number of SIPO output pins you are planning to use, i.e. virtual array output pin pool size = `number_SIPOs` x 8. Note that a value between 0 and 255 is permissible for `number_SIPOs`.

`number_timers` is the number of timers that the class will be configured for. Note that a value between 0 and 255 is permissible.

It recommended that these two class parameters are defined as macro values, thereby allowing them to be used throughout the sketch code, as required, and providing a degree of self documentation.

Examples:

1. `SIPO8 my_SIPOs(15, 4);`

But a better approach would be:

2.
```
#define number_SIPO_ICs  15
#define number_timers     4
SIPO8 my_SIPOs(number_SIPO_ICs, number_timers);
```

Step 3

Okay, we're off and running? Not quite, before we can plough on and start 'SIPOing' we need to create our banks. We need to allocate the SIPO output pins from the array pool that was created at class initiation to banks. These output pins will not be addressable and therefore active until they are assigned into banks.

We do this using the function `create_bank`. This function will create a new bank with the specified 3-wire interface of the required number of physical SIPO ICs. Again, use the data you have documented in the above template. Once a bank has been successfully created its associated pins become addressable (absolute and relative) and active. For example, we will create a bank using digital pins 8, 9 and 12 for our data, clock and latch signalling, respectively and of size 6 SIPO ICs:

```
int bank_id;
bank_id = my_SIPOs.create_bank(8,9,12, 6); // 6 SIPO ICs in bank
```

Things to notice:

- the `create_bank` function call is preceded with the name we have given to the **SIPO8** class, in this example '`my_SIPOs`'. This is required to access any resource within the class
- the function provides a return value of type `int`. If the creation is successful, this value is the reference you should use whenever you use any of the library resources where bank reference is a required parameter. How you retain this is very much up to your design, but see the example sketches below which should prove helpful. See Specifications – SIPO Bank Functions, `create_bank` to understand the possible return values/conditions.

The best place to create your banks is in the `setup()` function, but it can be done anywhere with the application of a little common sense.

Step 4

Now that is done we can start to use and drive our SIPO ICs. But before we do this recall that:

1. the output pins of physically connected SIPO ICs are referenced within a virtual pool of output pins
2. these virtual output pins are only active and therefore addressable when they belong to a bank
3. virtually active output pins can be accessed by absolute addressing or relative addressing, but using different methods
4. the status of virtual active pins will be either `LOW` or `HIGH`
5. physically connected SIPO output pins are not updated to reflect their virtual status unless and until a transfer or 'shift out' process is invoked, using one of the `xfer_...` functions.

For example:

```
int bank_id = my_SIPOs.create_bank(8,9,12, 1); // 1 SIPO ICs in bank
if (bank_id >= 0){
  // successfully created bank
  my_SIPOs.set_bank_SIPO(bank_id, 0, 0b01100111);
  // shift out to physical SIPO outputs
  my_SIPOs.xfer_bank(bank_id, MSBFIRST);
}
```

Additionally to note:

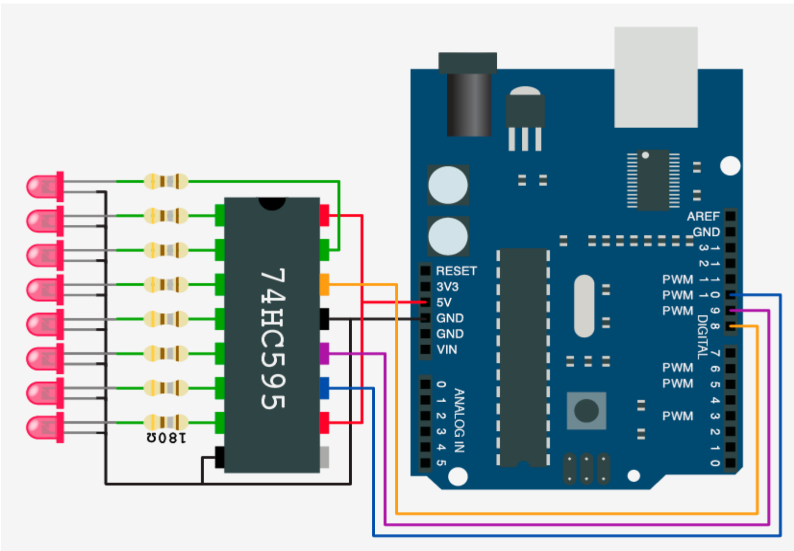
If you need to reference any of the library's class resources (<u>except macros definitions</u>) then you must prefix them with the name you have given to the class when you created/initiated it. For example, if we initiated the class with the name 'my_SIPOs' then referencing examples would look something like:

```
my_SIPOs.SIPO_banks[bank_id].bank_low_pin
my_SIPOs.SIPO_banks[2].bank_num_SIPOs
my_SIPOs.timers[timer3].timer_status
my_SIPOs.pin_status_bytes[4]
my_SIPOs.num_active_pins
my_SIPOs.num_pin_status_bytes
my_SIPOs.max_timers
etc.
```

See Specifications – Configuration Data & Variables.

Components and Basic Wiring Scheme for Single SIPO IC/Microcontroller

It is worthwhile putting together a basic test harness for use with this UG, the example sketches and the tutorials. The components (swap out the suggested 180 ohm resistors for 220 ohm resistors) and wiring scheme for the basic harness is:



Standard 74HC595/Arduino Wiring Scheme

Specifications – Configuration Data & Variables

Reserved Macro Definitions

Macro Definitions - #define	Values	Significance / Comments
pins_per_SIPO	8	The number of bits/output pins of a virtual/physical SIPO and on which the core design of the library is based.
create_bank_failure	-1	The return value from the create_bank function, if a request to create a new bank for the specified number of SIPOs cannot be met, e.g. too few remaining unallocated SIPOs in the unused pool.
pin_read_failure	-1	The return value from the read_array_pin & read_bank_pin functions, if a specified pin is not an active pin.
pin_invert_failure	-1	The return value from the invert_array_pin & invert_bank_pin functions, if a specified pin is not an active pin.
pin_set_failure	-1	The return value from the set_array_pin & set_bank_pin functions, if a specified pin is not an active pin.
bank_not_found	-1	The return value from the get_bank_from_pin, set_bank_SIPO, invert_bank_SIPO & read_bank_SIPO functions, if a specified pin/SIPO does not occupy a defined SIPO bank.
SIPO_not_found	-2	The return value from the set_bank_SIPO, invert_bank_SIPO & read_bank_SIPO functions, if a specified SIPO not in range for given bank.
timer0	0	Eight predefined macros are provided that may be used for 1 – 8 configured timers. Beyond eight, the end user will either reference timers explicitly, e.g. 9, 10, etc, or by using his/her own declared macros for such purposes.
timer1	1	
timer2	2	
timer3	3	
timer4	4	
timer5	5	
timer6	6	
timer7	7	
elapsed	true	Return value of the SIPO8_timer_elapsed function if the defined elapse period has completed for a specified timer.
not_elapsed	!elapsed	Return value of the SIPO8_timer_elapsed function if the defined elapse period for a specified timer has not yet completed for a specified timer.
active	true	Used internally by the SIPO8 timer functions to judge if a specified timer is active.
not_active	!active	Used internally by the SIPO8 timer functions to judge if a specified timer is inactive.

SIPO Control Structure (SCS)

Declarations/Variables	Purpose/definition
struct SIPO_control {	For each bank created maintains a record of:
uint8_t bank_data_pin;	Digital pin allocated to the data input line to a SIPO bank
uint8_t bank_clock_pin;	Digital pin allocated to the clock input line to a SIPO bank
uint8_t bank_latch_pin;	Digital pin allocated to the latch input line to a SIPO bank
uint8_t bank_num_SIPOs;	The number of SIPO ICs grouped under a bank
uint16_t bank_low_pin;	The first output pin number for the bank (absolute pin address)
uint16_t bank_high_pin;	The last output pin number for the bank (absolute pin address)
} *SIPO_banks;	SIPO_banks is the active/working data structure used throughout the library for managing and controlling SIPO data

Timer Control Structure (TCS)

Declarations/Variables	Purpose/definition
struct timer_control {	For each timer created maintains a record of:
bool timer_status;	current status of a timer - active or not_active
uint32_t start_time;	the millis time when a timer is started
} *timers;	timers is the active/working data structure used throughout the library for managing and controlling timers

Pin Status Bytes (PSB)

Declarations/Variables	Purpose/definition
uint8_t * pin_status_bytes;	An array sized and created to be the number of 8 bit unsigned bytes required to map the maximum number of SIPOs defined at class initiation. That is, one status byte per required SIPO IC. This structure is the virtual SIPO output pin map and represents the entire SIPO output pin array pool. The pin_status_bytes array is used to record the status (HIGH or LOW) of each physical SIPO output pin.

Other User Accessible Variables/Declarations

Declarations/Variables	Purpose/definition
`uint8_t max_SIPOs`	The maximum number of SIPO ICs defined at class initiation by the user code.
`uint16_t max_pins`	The maximum number of SIPO output pins defined for the SIPO pool. Note that this is not the number of active SIPO output pins, but is the maximum number of SIPO output pins available for allocation to SIPO banks via calls to the `create_bank` function. Therefore, `max_pins >= num_active_pins`.
`uint16_t num_active_pins`	The number of <u>active</u> SIPO output pins allocated from the SIPO output pin pool. Allocated by the `create_bank` function – see above.
`uint8_t num_pin_status_bytes`	The number of 8 bit unsigned bytes allocated to map the maximum number of SIPO output pins. For example, if the max number of SIPO output pins is 128, then this value will be 128/8 = 16 bytes.
`uint8_t num_banks`	The total number of SIPO banks created by using `create_bank` function.
`uint8_t bank_SIPO_count`	The total number of SIPO ICs in use by defined SIPO banks. Note that `bank_SIPO_count <= max_SIPOs`.
`uint8_t max_timers`	The number of timers defined at time of class initiation by the user code.

The above SIPO8 library variables are simply accessed in standard form, by prefixing the variable with '.class_name', -

```
<class name attributed by end user code>.<variable>
```

For example,

```
my_SIPOs.SIPO_banks[bank_id].bank_low_pin
my_SIPOs.SIPO_banks[2].bank_num_SIPOs
my_SIPOs.timers[timer3].timer_status
my_SIPOs.pin_status_bytes[4]
my_SIPOs.num_active_pins
my_SIPOs.num_pin_status_bytes
my_SIPOs.max_timers
etc.
```

See Steps to Successful Use, above.

Specifications - SIPO Functions

SIPO8 Class Constructor Function

<table>
<tr><td>Type</td><td>n/a</td><td>Name</td><td>SIPO8</td></tr>
<tr><td>Parameters</td><td colspan="3">(uint8_t Max_SIPO_ICs, uint8_t Max_timers)</td></tr>
<tr><td>Purpose / functionality</td><td colspan="3">Like all constructor functions its purpose is to establish a suitably configured environment within which the class can be applied and used by end user code. For the SIPO8 library, this involves the creation, from free memory, of all data structures and variables that properly describe and map the user's requirements.

The constructor has just two parameters – max_SIPO_ICs & Max_timers:

Max_SIPO_ICs parameter allows the SIPO environment to be sized to the maximum number of hardware 8bit SIPO ICs that will connected in some way or other. This parameter is also used to size and create the number of output pin status bytes required to map all pins of all SIPO output pins
Max_timers parameter specifies the number of timer data struct(ures) required for the end users needs
To note:
1. even though the environment is created to the number of SIPO ICs required, SIPO output pins are <u>not</u> addressable or usable unless and until they are allocated into banks
2. if the constructor function cannot established the environment, e.g. insufficient free memory to 'malloc', the SIPO_lib_exit function is called with a suitable error message, after which the code will force an exit with the specified reason code. See the function specification for SIPO_lib_exit
3. On successful setup, a number of key SIPO8 library variables are accessible to the end user, see Other User Accessible Variables/Declarations
4. the class constructor will allocate an amount of free memory dependent on the number of SIPO ICs and number of timers specified, as follow:

SIPO control structure 8 x Max_SIPO_ICs, bytes
SIPO status bytes 1 x Max_SIPO_ICs, bytes
SOPI timers 5 x Max_timers, bytes

For example, if we initiated the class as follows:

SIPO8 my_SIPOs(12,7); // 12 SIPO ICs, 7 timers

The class constructor will attempt to acquire (12 x 8) + (12 x 1) + (7 x 5) = 143 bytes of free memory.</td></tr>
<tr><td>Return values</td><td colspan="3">None</td></tr>
<tr><td colspan="4">Example(s)</td></tr>
<tr><td colspan="4">Example 1
SIPO8 my_SIPOs(12, 7);

Example 2
#define Max_SIPOs 12
#define Max_timers 7
SIPO8 my_SIPOs(Max_SIPOs, Max_timers);</td></tr>
</table>

SIPO Array Pool Functions (Absolute Addressing)

Type	int	Name	**set_all_array_pins**
Parameters	(bool pin_status)		
Purpose / functionality	The function sets every SIPO output pin in the active output pin array to the specified status. To note: 1. pin_status = LOW or HIGH 2. the function is equivalent to set_banks(pin_status)		
Return values	None		

Example(s)

Example 1:
```
...
my_SIPOs.set_all_array_pins(HIGH);  // all outputs to high
my_SIPOs.xfer_array(LSBFIRST); // shift out entire active array of SIPO pins
...
```

Example 2:
```
...
my_SIPOs.set_all_array_pins(LOW);  // all outputs to low
my_SIPOs.xfer_array(MSBFIRST); // shift out entire active array of SIPO pins
...
```

Type	int	Name	**invert_all_array_pins**
Parameters	none		
Purpose / functionality	The function inverts the existing status every SIPO output pin in the active output pin array. Note: 1. the function is equivalent to invert_banks().		
Return values	None		

Example(s)

Example 1:
```
...
my_SIPOs.invert_all_array_pins();  // invert all existing pin statuses
my_SIPOs.xfer_array(LSBFIRST); // shift out entire active array of SIPO pins
...
```

Example 2:
```
...
my_SIPOs.invert_all_array_pins();  // invert all existing pin statuses
my_SIPOs.xfer_array(MSBFIRST); // shift out entire active array of SIPO pins
...
```

<table>
<tr><td>Type</td><td></td><td>Name</td><td>set_array_pin</td></tr>
<tr><td>Parameters</td><td colspan="3"><code>(uint16_t pin, bool pin_status)</code></td></tr>
<tr><td>Purpose / functionality</td><td colspan="3">The function sets the specified SIPO output pin in the active output pin array to the specified status.

To note:
1. the <code>pin</code> parameter is an <u>absolute</u> pin address
2. <code>pin_status = LOW or HIGH</code></td></tr>
<tr><td>Return values</td><td colspan="3">The function will return one of two values:

1. the pin was successfully set, in which case the return value will be the pin address of the specified pin address parameter
2. the pin could not be set, for example the given pin address is not in the active range of pin addresses. In this case the return value is -1 (macro name of <code>pin_set_failure</code>)</td></tr>
<tr><td colspan="4">Example(s)</td></tr>
</table>

```
...
for (uint16_t pin = 17; pin < 93; pin++){
  if (my_SIPOs.set_array_pin(pin, LOW) == pin_set_failure){
    // set pin error
    ...
  }
}
...
```

<table>
<tr><td>Type</td><td><code>int</code></td><td>Name</td><td>invert_array_pin</td></tr>
<tr><td>Parameters</td><td colspan="3"><code>(uint16_t pin)</code></td></tr>
<tr><td>Purpose / functionality</td><td colspan="3">The function inverts the existing pin status of the specified SIPO output pin in the active output pin array.

To note:
1. the <code>pin</code> parameter is an <u>absolute</u> pin address
2. <code>pin_status = LOW or HIGH</code></td></tr>
<tr><td>Return values</td><td colspan="3">The function will return one of two conditions;

1. the specified pin was successfully inverted. In this case, the return value will be the updated output pin status value (<code>LOW</code> or <code>HIGH</code>)
2. the specified output pin could not be inverted, for example the pin address given is not in the active array. In this instance the return value will be -1 (macro name of <code>pin_invert_failure</code>)</td></tr>
<tr><td colspan="4">Example(s)</td></tr>
</table>

```
...
for (uint16_t pin = 32; pin < 48; pin++){
  my_SIPOs.invert_array_pin(pin);
}
...
```

Type		Name	**read_array_pin**
Parameters	(uint16_t pin)		
Purpose / functionality	The function will read the status of the specified output pin, returning its status or an error if not in the active range of the array. To note: 1. the pin parameter is an <u>absolute</u> pin address		
Return values	The function returns one of two conditions: 1. if successful, the function returns the pin's status – LOW or HIGH. 2. if unsuccessful, the function will return -1 (macro name of pin_read_failure)		

Example(s)

```
...
pin_status = my_SIPOs.read_array_pin(107); // pin 107 of the array
if (pin_status == pin_read_failure){
  // read error, pin 107 not in the active range of the array
  ...
}
...
```

Type	void	Name	**xfer_array**
Parameters	(bool msb_or_lsb)		
Purpose / functionality	This function transfers (xfers) / shifts out, from the virtual SIPO output pin array, all of the output pins within the active output pin array, with the direction of shift out determined by the msb_or_lsb parameter. To note: 1. the msb_or_lsb parameter is either LSBFIRST or MSBFIRST, least significant bit or most significant bit, respectively 2. the function is equivalent to xfer_banks()		
Return values	None		

Example(s)

```
...
my_SIPOs.set_all_array_pins(LOW);
my_SIPOs.xfer_array(LSBFIRST);
...
```

SIPO Bank Functions (Relative Addressing)

Type	int	Name	**create_bank**
Parameters	(uint8_t data_pin, uint8_t clock_pin, uint8_t latch_pin, uint8_t num_SIPOs)		
Purpose / functionality	This is an essential function for creating a bank of SIPO ICs (single or several in cascade form) and allocates output pins from the defined global array output pin pool. The bank control struct(ure) stores the given microcontroller 3-wire digital interface pins (data, clock and latch) used to drive the bank of SIPO ICs. The structure also records the absolute, inclusive, addresses of the first and last SIPO output pins associated with the bank.		
Return values	The function returns one of two conditions: 1. creation successful, in which case the return value is the bank address which should be used with any of the library's bank specific functions 2. creation unsuccessful, the return value is -1 (macro name of create_bank_failure)		

Example(s)

```
#define max_SIPOs  6
#define max_timers 4
#define num_banks  3

// initiate the class for max SIPOs/timers required
SIPO8 my_SIPOs(max_SIPOs, max_timers);
//
// bank_pins format - for each bank,
// [][0] = data pin number
// [][1] = clock pin number
// [][2] = latch pin number
// [][3] = number SIPOs in this bank
// [][4] = the bank number allocated for this bank by create_bank function
//
int bank_setup_data[num_banks][5] = {
  4, 3, 2,  2, -1,
  5, 6, 7,  3, -1,
  8, 9, 10, 1, -1
};
void setup() {
  Serial.begin(9600);
  for (uint8_t bank = 0; bank < num_banks; bank++) {
    bank_setup_data[bank][4] = my_SIPOs.create_bank(
                               bank_setup_data[bank][0], // data pin
                               bank_setup_data[bank][1], // clock pin
                               bank_setup_data[bank][2], // latch pin
                               bank_setup_data[bank][3]);// num SIPOs this bank
    if (bank_setup_data[bank][4] == create_bank_failure) {
      my_SIPOs.print_SIPO_data();
      Serial.println(F("failed to create bank"));
      Serial.flush();
      exit(0);
    }
  }
  my_SIPOs.print_SIPO_data();
}
...
```

Type	void	Name	**set_bank**
Parameters	(uint8_t bank, bool status)		
Purpose / functionality	The function will set every SIPO output pin defined by the bank to the given status. status is either LOW or HIGH.		
Return values	None		

Example(s)

```
...
my_SIPOs.set_bank(bank, HIGH);
...
```

Type	void	Name	**set_banks**
Parameters	(uint8_t from_bank, uint8_t to_bank, bool status)		
Purpose / functionality	For each bank in the from_bank/to_bank range of banks, the function will set every SIPO output pin defined by each bank to the given status. status is either LOW or HIGH.		
Return values	None		

Example(s)

```
...
my_SIPOs.set_banks(2, 5, LOW);
...
```

Type	void	Name	**set_banks**
Parameters	(bool status)		
Purpose / functionality	The function will set every SIPO output pin defined by every bank to the given status. Note: 1. status is either LOW or HIGH. 2. the function is equivalent to set_all_array_pins(status);		
Return values	None		

Example(s)

```
...
my_SIPOs.set_banks(LOW); // set all defined bank output pins to low
...
```

Type	void	Name	**invert_bank**
Parameters	(uint8_t bank)		
Purpose / functionality	The function will invert the existing status value of every SIPO output pin defined by the bank.		
Return values	None		

Example(s)

```
...
my_SIPOs.invert_bank(bank);
...
```

Type	void	Name	**invert_banks**
Parameters	(uint8_t from_bank, uint8_t to_bank)		
Purpose / functionality	For each bank in the from_bank/to_bank range of banks, the function will invert the existing output pin status of <u>every</u> SIPO output pin defined by <u>each</u> bank .		
Return values	None		

Example(s)

```
...
my_SIPOs.invert_banks(2, 5);
...
```

Type	void	Name	**invert_banks**
Parameters	none		
Purpose / functionality	The function will invert the existing status of every SIPO output pin of every bank. Note: 1. the function is equivalent to invert_all_array_pins();		
Return values	None		

Example(s)

```
...
my_SIPOs.invert_banks(); // invert output pins of all defined banks
...
```

Type	int	Name	**set_bank_pin**
Parameters	(uint8_t bank, uint8_t pin, bool status)		
Purpose / functionality	The function will set the specified SIPO output pin in the given bank to the given status. To note: 1. the pin address parameter is a relative address and <u>not</u> an absolute address. That is, regardless of the bank, the pin address will range from 0 to (8 x number of SIPO ICs in the bank) – 1. 2. the return value for a successful update will be the pin's absolute address 3. status is either LOW or HIGH.		
Return values	The function returns one of two conditions: 1. If successful, the function returns the pin's <u>absolute</u> address. 2. If unsuccessful, the function will return -1 (macro name of pin_set_failure). This will occur if the specified pin is out of range for the given bank.		

Example(s)

```
...
pin_abs_add = my_SIPOs.set_bank_pin(bank, 33, HIGH); // pin 33 of this bank
if (pin_abs_add == pin_set_failure){
  // set pin error, pin 33 not in range of this bank
  ...
}
...
```

Type	int	Name	**invert_bank_pin**
Parameters	(uint8_t bank, uint8_t pin)		
Purpose / functionality	The function will invert the existing status of the specified SIPO output pin in the given bank. To note: 1. the pin address parameter is a relative address and not an absolute address. That is, regardless of the bank, the pin address will range from 0 to (8 x number of SIPO ICs in the bank) – 1. 2. the return value for a successful update will be the pin's absolute.		
Return values	The function returns one of two conditions: 1. If successful, the function returns the pin's absolute address. 2. If unsuccessful, the function will return -1 (macro name of pin_invert_failure). This will occur if the specified pin is out of range for the given bank.		

Example(s)

```
...
pin_abs_add = my_SIPOs.invert_bank_pin(bank, 33, HIGH); // pin 33 of this bank
if (pin_abs_add == pin_invert_failure){
  // invert error, pin 33 not in range of this bank
  ...
}
...
```

Type	int	Name	**read_bank_pin**
Parameters	(uint8_t bank, uint8_t pin)		
Purpose / functionality	The function will read the existing status of the specified SIPO output pin in the given bank. To note: 1. the pin address parameter is a relative address and not an absolute address. That is, regardless of the bank, the pin address will range from 0 to (8 x number of SIPO ICs in the bank) – 1. 2. the return value for a successful update will be the pin's status – LOW or HIGH. This will occur if the specified pin is out of range for the given bank.		
Return values	The function returns one of two conditions: 1. If successful, the function returns the pin's status – LOW or HIGH. 2. If unsuccessful, the function will return -1 (macro name of pin_read_failure)		

Example(s)

```
...
pin_status = my_SIPOs.read_bank_pin(bank, 33); // pin 33 of this bank
if (pin_status == pin_read_failure){
  // read error, pin 33 not in range of this bank
  ...
}
...
```

Type	int	Name	**set_bank_SIPO**
Parameters	(uint8_t bank, uint8_t SIPO_num, uint8_t SIPO_value)		
Purpose / functionality	The specified SIPO within the given bank will be set to the 8bit value provided. (ie bank/SIPO = SIPO_value) To note: 1. the bank number is the bank id returned by the create_bank function when the bank is first created 2. the SIPO_num parameter is a relative value and is the number of the SIPO assigned to the given bank, starting at 0 3. the SIPO_value parameter is an unsigned 8bit value, representing each of the eight output SIPO pins 4. this function is equivalent to <u>eight</u> set_bank_pin calls which sets a single specified SIPO output pin. It can therefore a more efficient method to use if multiple output pins are to be set		
Return values	The function returns one of three conditions: 1. the given bank does not exist, the function will return -1 (macro name of bank_not_found) 2. the given bank does exist, but the given SIPO reference is not in range for the bank, the function will return -2 (macro name of SIPO_not_found) 3. the given bank <u>and</u> bank SIPO <u>do exist</u>, the function will return the address of the status byte (>=0) that is set with the given SIPO_value parameter		

Example(s)

Example 1

```
...
my_SIPOs.set_bank_SIPO(2, 0, 0b10101010); // bank 2, SIPO 0
my_SIPOs.set_bank_SIPO(2, 1, 0b11111111); // bank 2, SIPO 1
my_SIPOs.xfer_bank(2, MSBFIRST);
...
my_SIPOs.set_bank_SIPO(bank_id, 5,127); // SIPO 5 of current bank
my_SIPOs.xfer_bank(bank_id, LSBFIRST);
...
```

Example 2

```
#define num_patterns 4

uint8_t bit_patterns[num_patterns] = {
0b00001111,
0b11110000,
0b10101010,
0b01010101};
...
// set up banks 2 and 3 (4 SIPOs each) with initial and inverse
// patterns respectively
for (uint8_t pattern = 0; pattern < num_patterns; pattern++){
  my_SIPOs.set_bank_SIPO(2, pattern, bit_patterns[pattern]);
  my_SIPOs.set_bank_SIPO(3, pattern, ~bit_patterns[pattern]);
}
my_SIPOs.xfer_array(MSBFIRST);
...
```

Type	int	Name	**invert_bank_SIPO**
Parameters	(uint8_t bank, uint8_t SIPO_num)		
Purpose / functionality	This function will invert the existing 8bit SIPO output pin values. For example, if the existing values are 0b11110000, this function will set the output pins to 0b00001111, etc. To note: 1. the bank number is the bank id returned by the create_bank function when the bank is first created 2. the SIPO_num parameter is a relative value and is the number of the SIPO assigned to the given bank, starting at 0 3. this function is equivalent to <u>eight</u> invert_bank_pin calls which sets a single specified SIPO output pin. It can therefore a more efficient method to use if multiple output pins are to be set		
Return values	The function returns one of three conditions: 1. the given bank does not exist, the function will return -1 (macro name of bank_not_found) 2. the given bank does exist but the given SIPO reference is not in range for the bank, the function will return -2 (macro name of SIPO_not_found) 3. the given bank <u>and</u> bank SIPO <u>do exist</u>, the function will return the address of the status byte (>=0)		

Example(s)

```
...
my_SIPOs.invert_bank_SIPO(2, 0);        // invert bank 2, SIPO 0
my_SIPOs.invert_bank_SIPO(2, 1);        // invert bank 2, SIPO 1
my_SIPOs.xfer_bank(2, MSBFIRST);
...
my_SIPOs.invert_bank_SIPO(bank_id, 5); // invert SIPO 5 OF current bank
my_SIPOs.xfer_bank(bank_id, LSBFIRST);
...
```

<table>
<tr><td>Type</td><td>int</td><td>Name</td><td>read_bank_SIPO</td></tr>
<tr><td>Parameters</td><td colspan="3">(uint8_t bank, uint8_t SIPO_num)</td></tr>
<tr><td>Purpose / functionality</td><td colspan="3">This function will read the current status byte of the specified bank/SIPO. The returned value is the full eight bits representing the bank's SIPO pins.

To note:
1. the bank number is the bank id returned by the create_bank function when the bank is first created
2. the SIPO_num parameter is a relative value and is the number of the SIPO assigned to the given bank, starting at 0
3. this function is equivalent to <u>eight</u> read_bank_pin calls which reads a single specified SIPO output pin. It can therefore a more efficient method to use if multiple output pins are to be set</td></tr>
<tr><td>Return values</td><td colspan="3">The function returns one of three conditions:

1. the given bank does not exist, the function will return -1 (macro name of bank_not_found)
2. the given bank does exist but the given SIPO reference is not in range for the bank, the function will return -2 (macro name of SIPO_not_found)
3. the given bank <u>and</u> bank SIPO <u>do exist</u>, the function will return the 8bit value of the SIPO's output pins</td></tr>
<tr><td colspan="4">Example(s)</td></tr>
</table>

```
...
uint16_t status_byte = my_SIPOs.read_bank_SIPO(5,0);
if (status_byte >= 0){
  // valid read
  if (status_byte > 127){
    status_byte = status_byte - 127;
    my_SIPOs.set_bank_SIPO(5, 0, status_byte);
    my_SIPOs.xfer_bank(5, MSBFIRST);
  }
}
...
```

Other SIPO Functions

Type	int	Name	**get_bank_from_pin**
Parameters	(uint16_t pin)		
Purpose / functionality	The function returns the bank number in which the specified pin address resides. To note: 1. The specified pin address is an absolute address.		
Return values	The function returns one of two conditions: 1. If successful, the function returns bank number in which the pin resides. 2. If unsuccessful, the function will return -1 (macro name of bank_not_found). In this case the specified pin has not been allocated to a bank (via a create_bank call), albeit it may be defined within the pin array pool.		

Example(s)

```
...
for (uint16_t pin = 0; pin < my_SIPOs. num_active_pins; pin++) {
  Serial.print(pin);
  uint16_t bank = my_SIPOs.get_bank_from_pin(pin);
  if (bank == bank_not_found) {
    Serial.println(" - no bank for this pin");
  } else {
    Serial.print(" is in bank ");
    Serial.println(bank);
  }
    Serial.flush();
}
...
```

Type	int	Name	**num_pins_in_bank**
Parameters	(uint8_t bank)		
Purpose / functionality	The function returns the number of virtual and physical SIPO output pins the given bank maps/controls/owns. To note: 1. If the given bank is valid, the returned value is the number of output pins that the bank maps. The <u>relative</u> pin range for the given bank is therefore: `0, ..., my_SIPOs.num_bank_pins(bank) - 1`		
Return values	The function returns one of two conditions: 1. If successful, the function returns the number of SIPO output pins that the given bank maps 2. If unsuccessful, the function will return -1 (macro name of `bank_not_found`).		

Example(s)

```
...
int bank_id = my_SIPOs.get_bank_from_pin(pin);// determine which bank pin is in
if (bank_id != bank_not_found) {
  // pin belongs to an existing bank - determine how many pins in the bank
  uint16_t num_pins = my_SIPOs.num_pins_in_bank(bank_id);
  for (uint16_t rel_pin = 0; rel_pin < num_pins; rel_pin++){
    // process banks pins
    ...
 } else {
   Serial.print("\nbank not found for pin ");
   Serial.println(pin);
 }
...
```

Type	void	Name	**xfer_banks**
Parameters	(uint8_t from_bank, uint8_t to_bank, bool msb_or_lsb)		
Purpose / functionality	This function transfers (xfers) / shifts out, from the virtual SIPO output pin array, all of the output pins associated which each bank in turn, starting with `from_bank` and progressing to `to_bank`, with the direction of shift out determined by the `msb_or_lsb` parameter. To note: 1. the `msb_or_lsb` parameter is either `LSBFIRST` or `MSBFIRST`, least significant bit or most significant bit, respectively		
Return values	None		

Example(s)

```
...
my_SIPOs.xfer_banks(0,2,MSBFIRST);// transfers banks 0, 1 & 2 to physical SIPOs
...
```

Type	void	Name	**xfer_banks**
Parameters	(bool msb_or_lsb)		
Purpose / functionality	This function transfers (xfers) / shifts out, from the virtual SIPO output pin array, all of the output pins associated with <u>all</u> banks, with the direction of shift out determined by the msb_or_lsb parameter. To note: 1. the msb_or_lsb parameter is either LSBFIRST or MSBFIRST, least significant bit or most significant bit, respectively 2. the function is equivalent to xfer_array()		
Return values	None		

Example(s)

```
...
my_SIPOs.xfer_banks(MSBFIRST);// transfers all banks to the physical SIPOs
...
```

Type	void	Name	**xfer_bank**
Parameters	(uint8_t bank, bool msb_or_lsb)		
Purpose / functionality	This function transfers (xfers) / shifts out, from the virtual SIPO output pin array, all of the output pins associated with the specified bank, with the direction of shift out determined by the msb_or_lsb parameter. To note: 1. the msb_or_lsb parameter is either LSBFIRST or MSBFIRST, least significant bit or most significant bit, respectively		
Return values	None		

Example(s)

```
...
//strobe example, strobe bank 0 pins forward and back
my_SIPOs.set_bank(0, LOW); // clear down all pins in the bank
bool msb_or_lsb   = LSBFIRST;
// extract low and high absolute pin addresses from the bank data
uint16_t low_pin  = my_SIPOs.SIPO_banks[0].bank_low_pin;
uint16_t high_pin = my_SIPOs.SIPO_banks[0].bank_high_pin;
do {
 for (uint16_t pin = low_pin; pin <= high_pin; pin++) {
   my_SIPOs.set_array_pin(pin, HIGH);// as we know the pin's absolute address
   my_SIPOs.xfer_bank(0, msb_or_lsb);// xfer out the bank pins
   delay(100);
   my_SIPOs.set_array_pin(pin, LOW);
   my_SIPOs.xfer_bank(0, msb_or_lsb);
 }
   msb_or_lsb = !msb_or_lsb; // invert the direction of pin xfer
} while (true);
...
```

Type	void	Name	**print_pin_statuses**
Parameters	()		
Purpose / functionality	This function prints to the serial monitor the status of all output pins active/defined in the array. To note: 1. the output is show bank by bank, showing the status of each pin owned/defined by a bank 2. status bytes are printed, bit by bit, in standard convention starting with the most significant bit (MS) 3. the example sketch snippet sets up 104 SIPO pins allocated into four differently sized banks.		
Return values	None		

Example(s)

```
...
#define Max_SIPOs  13
#define Max_timers  4
#define Num_banks   4

// initiate the class for max SIPOs/timers required
SIPO8 my_SIPOs(Max_SIPOs, Max_timers);

//
// bank_pins format - for each bank,
// [][0] = data pin number
// [][1] = clock pin number
// [][2] = latch pin number
// [][3] = number SIPOs in this bank
// [][4] = the bank number allocated for this bank by create_bank function
//
int bank_setup_data[Num_banks][5] = {
   4,  3,  2, 1, -1,
   5,  6,  7, 2, -1,
   8,  9, 10, 6, -1,
  11, 12, 13, 4, -1
};
void setup() {
  Serial.begin(9600);
  for (uint8_t bank = 0; bank < Num_banks; bank++) {
    bank_setup_data[bank][4] = my_SIPOs.create_bank(
                                 bank_setup_data[bank][0],
                                 bank_setup_data[bank][1],
                                 bank_setup_data[bank][2],
                                 bank_setup_data[bank][3]);
    if (bank_setup_data[bank][4] == create_bank_failure) {
      my_SIPOs.print_SIPO_data();
      Serial.println(F("failed to create bank"));
      Serial.flush();
      exit(0);
    }
  }
  my_SIPOs.print_SIPO_data();  // display SIPO data set up
  randomSeed(analogRead(A0));
  for (uint8_t pin = 0; pin < my_SIPOs.num_active_pins; pin++) {
    bool pin_status = random(LOW, HIGH + 1);
    my_SIPOs.set_array_pin(pin, pin_status);
  }
  my_SIPOs.print_pin_statuses(); // display status of all active pins
...
```

Type	void	Name	**print_pin_statuses**

Example output from the above sketch snippet:

```
SIPO global values:
pins_per_SIPO    = 8
max_SIPOs        = 13
bank_SIPO_count = 13
num_active_pins = 104
num_pin_status_bytes = 13
next_free bank =  all SIPOs used
Number timers   =  4

Bank data:
bank = 0
  num SIPOs =      1
  latch_pin =      2  clock_pin =     3  data_pin   =    4
  low_pin   =      0  high_pin  =     7
bank = 1
  num SIPOs =      2
  latch_pin =      7  clock_pin =     6  data_pin   =    5
  low_pin   =      8  high_pin  =     23
bank = 2
  num SIPOs =      6
  latch_pin =      10  clock_pin =    9  data_pin   =    8
  low_pin   =      24  high_pin  =    71
bank = 3
  num SIPOs =      4
  latch_pin =      13  clock_pin =    12  data_pin   =   11
  low_pin   =      72  high_pin  =    103

Active pin array, pin statuses:
Bank  0: MS01110001LS
Bank  1: MS1011110111100111LS
Bank  2: MS011011111011111011101100011011001100010001010010LS
Bank  3: MS00011001000010111010000010001101LS
```

Type	`void`	Name	**print_SIPO_data**
Parameters	`()`		
Purpose / functionality	The function prints to the serial monitor the configuration data defined by the end user setup. This is a very useful function to use during development and debugging, and should be removed once the code is robust.		
Return values	None		

Example(s)

The follow SIPO data is shown for a class initiation setup of:

```
#define max_SIPOs  7
#define max_timers 4
#define num_banks  3

// initiate the class for max SIPOs/timers required
SIPO8 my_SIPOs(max_SIPOs, max_timers);
```

Output from a print_SIPO_data call following the above setup:

```
SIPO global values:
pins_per_SIPO   = 8
max_SIPOs       = 7
bank_SIPO_count = 7
num_active_pins = 56
num_pin_status_bytes = 7
next_free bank =  all SIPOs used
Number timers   =  4

Bank data:
bank = 0
  num SIPOs =      2
  latch_pin =      2  clock_pin =     3  data_pin  =     4
  low_pin   =      0  high_pin  =     15
bank = 1
  num SIPOs =      1
  latch_pin =      7  clock_pin =     6  data_pin  =     5
  low_pin   =      16  high_pin  =    23
bank = 2
  num SIPOs =      4
  latch_pin =      10  clock_pin =    9  data_pin  =     8
  low_pin   =      24  high_pin  =    55
```

<table>
<tr><td>Type</td><td>int</td><td>Name</td><td>SIPO8_timer_elapsed</td></tr>
<tr><td>Parameters</td><td colspan="3">(uint8_t timer, uint32_t elapsed_time)</td></tr>
<tr><td>Purpose / functionality</td><td colspan="3">This function determines if the specified timer has completed the elapsed_time parameter count.

To note:
1. the elaspsed_time parameter is the number of milliseconds to elapse</td></tr>
<tr><td>Return values</td><td colspan="3">The function returns the follow values:

1. if the timer is <u>active</u> and <u>has</u> elapsed, the return value is true (macro name of elapsed)
2. if the timer is <u>active</u> and has <u>not</u> elapsed, the return value is false (macro name of not_elapsed)
3. if the timer is <u>not</u> active the function will return a value of false (macro name of not_elapsed)</td></tr>
<tr><td colspan="4">Example(s)</td></tr>
</table>

```
...
//flash LED pin on pins 0 and 3, half cycle of 1 sec and 2 secs, respectively
  uint8_t pin0 = 0;
  uint8_t pin3 = 3;
  my_SIPOs.SIPO8_start_timer(timer0);
  my_SIPOs.SIPO8_start_timer(timer1);
  do {
    if (my_SIPOs.SIPO8_timer_elapsed(timer0, 1000) == elapsed) {
      my_SIPOs.invert_array_pin(pin0);
      my_SIPOs.xfer_bank(0, MSBFIRST);
      my_SIPOs.SIPO8_start_timer(timer0);
    }
    if (my_SIPOs.SIPO8_timer_elapsed(timer1, 2000) == elapsed) {
      my_SIPOs.invert_bank_pin(0, pin3);
      my_SIPOs.xfer_bank(0, MSBFIRST);
      my_SIPOs.SIPO8_start_timer(timer1);
    }
  } while (true);
...
```

Type	void	Name	**SIPO8_start_timer**
Parameters	(uint8_t timer)		
Purpose / functionality	This function will start an elapsing timer for the given timer parameter. As many timers may be used as defined as class initiation. To note: 1. the library maintains eight macros that can be used by end user code, these being `timer0, timer1, ... , timer7`. For example if we wish to start timer 3, the call would be `my_SIPOs.SIPO8_start_timer(timer3)` 2. the function marks the specified timer as active and records the `millis` time as the start of the elapse time count 3. if the timer parameter is not valid, i.e. not in range of the defined number of timers, the call is ignored		
Return values	None		

Example(s)

example 1:

```
...
// start all defined timers
for (uint8_t timer = 0; timer < max_timers; timer++){
  my_SIPOs.SIPO8_start_timer(timer);
}
...
```

example 2:

```
...
my_SIPOs.SIPO8_start_timer(timer0);
my_SIPOs.SIPO8_start_timer(timer4);
my_SIPOs.SIPO8_start_timer(timer7);
...
```

Type	void	Name	**SIPO8_stop_timer**
Parameters	(uint8_t timer)		
Purpose / functionality	This function stops an elapsing timer for the given timer parameter. To note: 1. the library maintains eight macros that can be used by end user code, these being `timer0, timer1, ... , timer7`. For example if we wish to stop timer 6, the call would be `my_SIPOs.SIPO8_stop_timer(timer6)` 2. the function marks the specified timer as inactive 3. if the timer parameter is not valid, i.e. not in range of the defined number of timers, the call is ignored.		
Return values			
Example(s)			

example 1:

```
...
// stop all defined timers
for (uint8_t timer = 0; timer < max_timers; timer++){
  my_SIPOs.SIPO8_stop_timer(timer);
}
...
```

example 2:

```
...
my_SIPOs.SIPO8_stop_timer(timer0);
my_SIPOs.SIPO8_stop_timer(timer4);
my_SIPOs.SIPO8_stop_timer(timer7);
...
```

Private Functions

Type	void	Name	**shift_out_bank**
Parameters	(uint8_t data_pin, uint8_t clock_pin, uint8_t status_bits, bool msb_or_lsb)		
Purpose / functionality	The function shifts out (xfers) the given byte value parameter `status_bits`, to the associated 3-wire SIPO serial interface in the specified direction, determined by the `msb_or_lsb` parameter. Note that this is a private function and is not accessible by end user code. To note: 1. the `msb_or_lsb` parameter is either `LSBFIRST` or `MSBFIRST`, least significant bit or most significant bit, respectively		
Return values	None		

Example(s)

```
...
 digitalWrite(latch_pin, LOW); //  tell IC data transfer to start
 for (uint8_t SIPO = 0; SIPO < num_SIPOs_this_bank; SIPO++) {
   if (msb_or_lsb == LSBFIRST) {
     SIPO_status_byte = SIPO_first_status_byte + SIPO;
   } else {
     SIPO_status_byte = SIPO_last_status_byte - SIPO;
   }
 uint8_t status_bits = pin_status_bytes[SIPO_status_byte];
 shift_out_bank(data_pin, clock_pin, status_bits, msb_or_lsb);
 }
 digitalWrite(latch_pin, HIGH); //  tell IC data transfer is finished
...
```

Type	void	Name	**SIPO_lib_exit**
Parameters	(uint8_t reason)		
Purpose / functionality	This is a private function and is called during the class initiation by the class constructor function if the required memory cannot be acquired for the library's data structures. The function is not accessible by end user code. If an error is found, then alert output is directed to the serial monitor, set for 9600 baud during your development and testing. The possible error conditions/text are: Reason 0 – insufficient free memory to establish the SIPO banks : "`Exit:out of memory for setup-SIPO banks`" Reason 1 – insufficient free memory to establish the SIPO status pin bytes : "`Exit:out of memory for setup-status bytes`" Reason 2 – insufficient free memory to establish timer struct(ure) : "`Exit:out of memory for setup-timers`" Default reason – any termination reason not 0, 1 or 2: "`Exit:unspecified`" In all cases, the program will terminate with a `reason` exit code.		
Return values	None		

Example(s)

```
SIPO_lib_exit(1); // cant map status bytes
```

Corollary

As the SIPO8 library developed and grew to fulfil its design objectives, it became clear that it was able to also support new features not foreseen at the outset. One such unforeseen feature is SIPO bank interleaving.

SIPO bank interleaving allows us to define any number of banks (of the same size, i.e. the number of physical SIPOs in a bank) to the same physical 3-wire microcontroller digital interface. In this way we can maintain a number of different SIPO bank output pin configurations for the same physical SIPO bank and xfer each or any of these independently as appropriate for our purposes.

For a simple example of SIPO bank interleaving see Tutorial 2.

Tutorials

In addition to the example sketches provided below, a number of tutorials have been developed to further aid understanding in how the SIPO8 library may be utilised. The tutorials build one on another so that by the end of the full course the reader should be well armed to use the library in depth and with great effect. If you are wish to start at more sedate pace then look at the tutorials before tackling the examples below.

Tutorial	Title	Theme/Topic
Tutorial 1	Setup & Absolute Addressing	Introduces how to set up the library's class and covers absolute addressing of SIPO outputs, using a single SIPO IC
Tutorial 2	Relative Addressing	Introduces relative addressing of SIPO outputs through the construct of SIPO banks, using a single SIPO IC
Tutorial 3	Using Timers	Covers the principles of using the SIPO8 library timer capabilities, again using a single SIPO IC
Tutorial 4	Cascading SIPOs	Guides the reader through creating a two SIPO bank in a cascade. The principles covered then allow SIPO banks of any size to be created through straight forward extension of SIPO ICs but with very little change to the driving software sketch
Tutorial 5	Bank Interleaving	The tutorial looks at the feature of bank interleaving where it is possible to map any number of SIPO banks of the same size to the same 3-wire digital interface, bringing banks 'into play' as required.
Tutorial 6	Q & A	This tutorial provides answers to specific questions concerning "how do I..", "how can I…", etc.

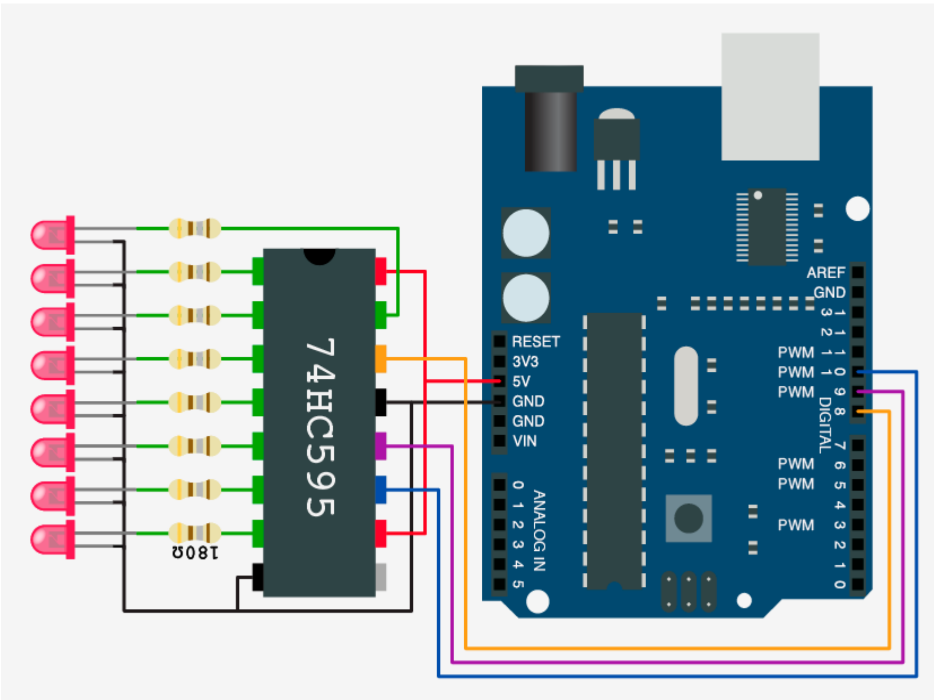
Example Sketches

The following example sketches help to illustrate some of the basic features of the SIPO8 library. The sketches may be downloaded from github by following the links with each example sketch.

In summary, the example sketches demonstrate use of the library to:

- Flash eight LEDs at independent frequecies using SIPO8 library timers, employing relative output pin addressing within a bank
- Strobe a bank of LEDs, employing absolute output pin addressing
- Chasing LEDs, as might be seen on a clock dial, running around its circumference, with each LED indication the passing of one second
- Drive a 7 segment LED matrix display, highlighting the different shift out directions

Before jumpling straight into the examples, prepare the basic components needed for each example. The components (swap out the suggested 180 ohm resistors for 220 ohm resistors) and wiring scheme for the first three exmple sketches are:



Standard 74HC595/Arduino Wiring Scheme

For the final example, 7 segment LED matrix display, use the same layout as above but remove the LEDs. See the example for pin connections between the 74HC595 and the 7 segment LED matrix.

Sketch Example – Flashing LEDs, Absolute Addressing

In this sketch we demonstrate absolute addressing functions to update the SIPO virtual output pin array, having firstly created a single SIPO bank as per definitions below. Each physical SIPO output pin will have a LED connected and will be associated with its own SIPO8 timer so that it may be flashed with an independent frequency. Further use of the SIPO8 timer features is also made by running the sketch for a predefine period of time.

Note that the following changes may be made as desired:

1. The frequency of each of the eight LEDs may be independently varied in the presets of the array 'timer_intervals'
2. The order in which LEDs are processed may be made by changing the order of the presets in the array 'timer_pins'
3. The time for which the demonstration runs may be varied by altering the macro definition '#define demo_time' to any period in milliseconds.

Project:	Sketch Example – Flashing LEDs, Absolute Addressing						Date:	April 2021
User Defined Parameters				create_bank Results			Comments	
Control Pins			Num SIPO ICs in Bank	Bank Num	First Pin	Last Pin		
Data	Clock	Latch						
8	10	9	1	0	0	7	Eight connected LEDs, one to each SIPO pin and each LED will having its own timer	

Download this example from github.

The global SIPO8 data for this sketch are:

```
SPIC global values:
pins_per_SPIC    = 8
max_SPICs        = 1
bank_SPIC_count = 1
num_active_pins = 8
num_pin_status_bytes = 1
next_free bank =  all SPICs used
Number timers   =  9

Bank data:
bank = 0
  num SPICs  =1
  latch_pin  =9  clock_pin =      10  data_pin  =     8
  low_pin    =0  high_pin  =      7
```

```
//
//   Flash LEDs -
//   This sketch independently flashes eight LEDs each connected to a SIPO o/p pin,
//   each at a different frequency, using SIPO8 library timers.
//   The demonstration runs for a fixed time before terminating, again this being
//   controlled by a SIPO8 library timer.
//
//   This example uses absolute array pin addressing.
//
//   Ron D Bentley, Stafford, UK
//   April 2021
//
//   This example and code is in the public domain and
```

```
//   may be used without restriction and without warranty.
//
#include "ez_SPIC8_lib.h"
#define Max_SPICs   1
#define Max_timers 9 // we will create 9 timers, 0-7 for LEDs and 8 for our timing
#define my_timer   8 // we will use SPIC8 timer 8 for timing our demonstration

#define number_banks     1 // demonstration using 1 bank
#define demo_time    20000 // time in milliseconds flash demonstration to run for

int bank0_id;

// setup pin/LED flash data
uint8_t timer;

uint32_t timer_intervals[Max_timers] = {
// millisecond elapse timer values for each timer, 0 - 7
  300, 400, 500, 600, 700, 800, 900, 1000
};

uint8_t timer_pins[Max_timers] = {
  // SPIC output pin absolute addresses for timers 0 - 7
  0, 1, 2, 3, 4, 5, 6, 7
};

SPIC8 my_SPICs(Max_SPICs, Max_timers); // initiate the class for max SPICs/timers

void setup() {
  Serial.begin(9600);
  // create a single bank, params are:
  // data pin, clock pin, latch pin, number of SPICs this bank
  bank0_id = my_SPICs.create_bank(8, 10, 9, number_banks);
  if (bank0_id == create_bank_failure) {
    Serial.println(F("\nfailed to create bank"));
    Serial.flush();
    exit(0);
  }
  // print the bank data for confirmation/inspection
  my_SPICs.print_SPIC_data();
}
void loop() {
  // now lets add in the flashing LED code from Tutorial 3
  // start by setting all SPIC outputs to low (off)
  my_SPICs.set_all_array_pins(LOW);// set all declared virtual output pins LOW/off
  my_SPICs.xfer_array(LSBFIRST);    // move virtual pins to real SPIC output pins
  my_SPICs.SPIC8_start_timer(my_timer);  // start my_timer
  // keep processing until our my_timer has elapsed
  // start all timers
  for (timer = 0; timer < Max_timers - 1; timer++) {
    my_SPICs.SPIC8_start_timer(timer);
  }
  timer = 0; // start checking at first timer
  do {
    // check each timer for elapse and, if elapsed, invert the timer's output pin
    // and reset the timer
    if (my_SPICs.SPIC8_timer_elapsed(timer, timer_intervals[timer]) == elapsed) {
      my_SPICs.invert_array_pin(timer_pins[timer]); // invert the pin associated
with this timer
      my_SPICs.xfer_array(MSBFIRST);         // update physical SPIC outputs
      my_SPICs.SPIC8_start_timer(timer);     // reset/restart the current timer
    }
    timer++;  // move on to next timer
    if (timer >= Max_timers - 1) timer = 0; // wrap around to beginning
  } while (my_SPICs.SPIC8_timer_elapsed(my_timer, demo_time) != elapsed);
  //
  // end of LED flash demonstration - now clear down LEDs to off(LOW)
  my_SPICs.set_all_array_pins(LOW);// set all declared virtual output pins LOW/off
  my_SPICs.xfer_array(LSBFIRST);    // move virtual pin statuses to real SPIC output
pins
  exit(0);
}
```

Sketch Example – Strobing LEDs, Relative Addressing

In this sketch we demonstrate relative addressing functions to update the SIPO virtual output pin array, having firstly created a single SIPO bank as per definitions below. Each physical SIPO output pin will have a LED connected and the entire bank will be strobed at the given frequency.

Note that the following changes may be made as desired:

1. The frequency of the SIPO bank strobe is determined by the macro '`#define strobe_frequency`'. This may be varied for different rates of strobing

Project:	Sketch Example – Strobing LEDs, Relative Addressing						Date:	April 2021
User Defined Parameters				**create_bank** Results			Comments	
Control Pins			Num SIPO ICs in Bank	Bank Num	First Pin	Last Pin		
Data	Clock	Latch						
8	10	9	1	0	0	7	Eight connected LEDs, one per SIPO pin	

Download this example from github.

The global SIPO8 data for this sketch are:

```
SIPO global values:
pins_per_SIPO    = 8
max_SIPOs        = 1
bank_SIPO_count = 1
num_active_pins = 8
num_pin_status_bytes = 1
next_free bank =  all SIPOs used
Number timers   =  0

Bank data:
bank = 0
  num SIPOs =1
  latch_pin =9  clock_pin =      10  data_pin  =     8
  low_pin   =0  high_pin  =      7
```

```
//
//   Strobe -
//   This sketch strobes a number of LEDs driven by a SIPO IC, 8 output pins
//   forwards and backwards at a defined frequency.
//
//   This example uses relative bank addressing.
//
//   Ron D Bentley, Stafford, UK
//   April 2021
//
//   This example and code is in the public domain and
//   may be used without restriction and without warranty.
//

#include <ez_SIPO8_lib.h>

#define Max_SIPOs  1 // 1 x SIPOs - provides 8 output pins
#define Max_timers 0 // no timers required, will use delay

#define data_pin    8
#define clock_pin  10
#define latch_pin   9

#define strobe_frequency     50  // milliseconds, delay frequency
```

```
// initiate the class for max SIPOs/timers required
SIPO8 my_SIPOs(Max_SIPOs, Max_timers);

int bank_id;  // used to keep the SIPO bank id

void setup() {
  Serial.begin(9600);
  // create a bank of 'Max_SIPOs' using create_bank function:
  bank_id = my_SIPOs.create_bank(data_pin, clock_pin, latch_pin, Max_SIPOs);
  if (bank_id == create_bank_failure) {
    Serial.println(F("\nfailed to create bank, terminated"));
    Serial.flush();
    exit(0);
  }
  // start by setting all SIPO outputs to low (off) in the SIPO bank
  my_SIPOs.set_bank_SIPO(bank_id, 0, 0b00000000); // only 1 SIPO in bank,, index 0
  my_SIPOs.xfer_bank(bank_id, MSBFIRST);
  // print the bank data for confirmation/inspection
  my_SIPOs.print_SIPO_data(); }

void loop() {
  //strobe example, strobe bank_id pins forward and back
  bool msb_or_lsb = MSBFIRST; // pick an initial strobe direction
  int pins_in_bank = my_SIPOs.num_pins_in_bank(bank_id); // number SIPO output pins
in this bank
  if (pins_in_bank > 0) {
    do {
      for (uint16_t pin = 0; pin < pins_in_bank; pin++) {
        my_SIPOs.set_bank_pin(bank_id, pin, HIGH); // set next strobe pin
        my_SIPOs.xfer_bank(bank_id, msb_or_lsb);   // update physical SIPO bank
        delay(strobe_frequency);
        my_SIPOs.set_bank_pin(bank_id, pin, LOW);  // clear next strobe pin
        my_SIPOs.xfer_bank(bank_id, msb_or_lsb);   // update physical SIPO bank
      }
      msb_or_lsb = !msb_or_lsb; // switch strobe direction
    } while (true);
  } else {
    Serial.println(F("\nbank not found, terminated"));
    Serial.flush();
    exit(0);
  }
}
```

Sketch Example – Clock Chasing Second LEDs

In this sketch we demonstrate a chasing LED effect to represent the passing of each second on a clock face, with one LED representing one second. Hence the sketch will be designed with eight SIPO ICs, giving a total of 64 output pins, although we will limit the sketch to the first 60 of these. Each LED will be illuminated for each passing second until all 60 have been lit (one minute), after which all LEDs will be extinguished and the cycle restarted.

Although the sketch is initially designed as a clock second chaser, it may be adjusted for the number of LEDs chased (up to 64 without adding more SIPO ICs) and the frequency at which LED illumination occurs. To make these changes vary the macro definitions:

1. '`#define chase_length`' – set this from 0 – 64 output pins
2. '`#define frequency`' – set this to be the frequency at which LED illumination is to occur

Note that this example uses absolute addressing functions. As an exercise, make changes to use relative addressing functions (see previous sketch example - Strobing LEDs, Relative Addressing).

Project:	Sketch Example – Clock Chasing Second LEDs						Date:	April 2021
User Defined Parameters				**create_bank** Results			Comments	
Control Pins			Num SIPO ICs in Bank	Bank Num	First Pin	Last Pin		
Data	Clock	Latch						
8	10	9	8	0	0	63	Eight SIPO ICs in a single bank with LEDs connected to first 60 output pins. Timing driven by SIPO8 timer0.	

Download this example from github.

The global SIPO8 data for this sketch are:

```
SIPO global values:
pins_per_SIPO    = 8
max_SIPOs        = 8
bank_SIPO_count = 8
num_active_pins = 64
num_pin_status_bytes = 8
next_free bank =  all SIPOs used
Number timers   =  1

Bank data:
bank = 0
  num SIPOs  = 8
  latch_pin  = 9   clock_pin =       10   data_pin   =      8
  low_pin    = 0   high_pin  =       63
```

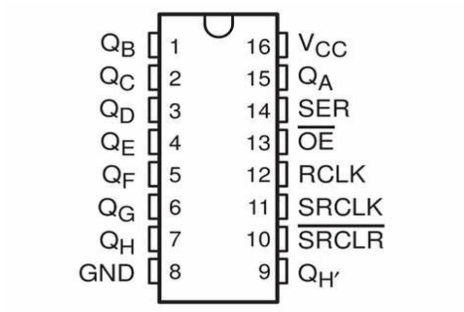
```
//
//   Chasing LEDs -
//   This sketch illuminates a defined sequential array of LEDs connected to
//   SIPO output pins, at a specified frequency (milliseconds).
//   At the end of the LED sequence, the LEDs are reset and the
//   cycle repeated.
//
//   The sketch can be configured for a SIPO array of any length, but this
//   example is configured to demonstrate the chasing LEDs of a clock dial.
//   In this instance the SIPO array length will be 64 output pins (8 x SIPOs)
//   but only the first 60 outputs will be used (0-59 seconds). The frequency
//   will be set to 1 second (1000 millisecs).
//
//   This example uses absolute addressing of the defined SIPO array.
//
//   Ron D Bentley, Stafford, UK
//   April 2021
//
//   This example and code is in the public domain and
//   may be used without restriction and without warranty.
//
#include <ez_SIPO8_lib.h>

#define Max_SIPOs  8 // 8 x SIPOs - provides 64 output pins
#define Max_timers 1 // used to count 1 second elapsing 'beats'

#define data_pin    8
#define clock_pin  10
#define latch_pin   9

#define chase_length  60   // chasing seconds on a clock
#define frequency     1000 // milli seconds - 1 second frequency

// initiate the class for max SIPOs/timers required
SIPO8 my_SIPOs(Max_SIPOs, Max_timers);

int my_LEDs;  // used to keep the SIPO bank id

void setup() {
  Serial.begin(9600);
  // create a bank of 'Max_SIPOs' using create_bank function:
  my_LEDs = my_SIPOs.create_bank(data_pin, clock_pin, latch_pin, Max_SIPOs);
  if (my_LEDs == create_bank_failure) {
    Serial.println(F("\nfailed to create bank"));
    Serial.flush();
    exit(0);
  }
  // start by setting all SIPO outputs to low (off)
  my_SIPOs.set_all_array_pins(LOW);
  my_SIPOs.xfer_array(MSBFIRST);
  // print the bank data for confirmation/inspection
  my_SIPOs.print_SIPO_data();
}
void loop() {
  uint8_t  next_pin = 0;
  my_SIPOs.SIPO8_start_timer(timer0); // kick off the timer
  do {
    if (my_SIPOs.SIPO8_timer_elapsed(timer0, frequency) == elapsed) {
      // 1 second time elapsed, so update next pin in the array
      my_SIPOs.SIPO8_start_timer(timer0); // restart 1 second count for next cycle
      if (next_pin == chase_length) {     // wrap around
        my_SIPOs.set_all_array_pins(LOW); // clear all pins
        next_pin = 0;
      } else {
        my_SIPOs.set_array_pin(next_pin, HIGH); // set absolute next_pin pin status
        next_pin++;
      }
      my_SIPOs.xfer_array(MSBFIRST); // update physical SIPOs
    }
  } while (true);
}
```

Sketch Example – 7 Segment LED Matrix

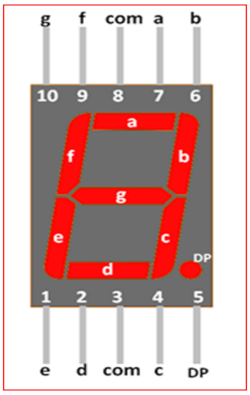
We finish our example sketches with a look at connecting and driving a 7 segment LED matrix display, common cathode, from a single SIPO IC (74HC595). As a twist, and for your future reference and use, the example drives a 7 segment LED matrix in two directions, MSBFIRST and LSBFIRST. This requires two different character bit patterns definitions, defining each of the 17 characters that the display will support (0 to hex F, plus DP ".").

The character bit patterns are stored as unsigned byte arrays, one representing the MSBFIRST character patterns and the other the LSBFIRST character patterns.

The pin outs for the SIPO IC (74HC595) and the 7 segment LED matrix are:



74HC595 IC



7 Segment LED Matrix

This example uses the same basic components/wiring scheme as for the previous examples except that the LEDs are removed and connection made to the 7 segment LED matrix as per table below. Note that each pin of the matrix connects to a corresponding pin of the 74HC595 IC, but after each of the 220 ohm resistors. Position the 7 segment LED matrix conveniently on your breadboard to aid wiring.

The pin to pin connections are:

Project:	Sketch Example – 7 LED Matrix						Date: April 2021
User Defined Parameters				create_bank Results			Comments
Control Pins			Num SIPO ICs in Bank	Bank Num	First Pin	Last Pin	
Data	Clock	Latch					
8	10	9	1	0	0	7	Pin mappings 74HC595 Matrix Q0/resistor→ 7 Q1/resistor→ 6 Q2 /resistor→ 4 Q3/resistor→ 2 Q4/resistor→ 1 Q5 /resistor→ 9 Q6/resistor→ 10 Q7/resistor→ 5 Connect after each 220 ohm resistor to each of the 7 segment LED matrix pins as above.

Download this example from github.

The global SIPO8 data for this sketch are:

```
SIPO global values:
pins_per_SIPO    = 8
max_SIPOs        = 1
bank_SIPO_count  = 1
num_active_pins  = 8
num_pin_status_bytes = 1
next_free bank =  all SIPOs used
Number timers   =  0

Bank data:
bank = 0
  num SIPOs  =1
  latch_pin  =9  clock_pin =      10  data_pin  =      8
  low_pin    =0  high_pin  =      7
```

```
//
//   7 Segment LED Matrix -
//   Sketch uses a single SIPO IC to map the 7 segment matrix and creates
//   a single bank comprising one SIPO IC.
//
//   This sketch drives a single 7 segment LED matrix, displaying digits
//   from 0 to hex F, in two repeating cycles:
//   1. cycle 1 - shifting out the most significant bit first (MSBFIRST), with each
//      character appended with the DP character ".", eg "3."
//   2. cycle 2 - shifting out the least significant bit first (LSBFIRST), without
//      the DP character appended.
//
//   The sketch offers two methods for updating a 7 Segment LED Matrix, choice
//   is a simple matter a preference/familiarity.
//
//   This example uses relative bank addressing.
//
//   Ron D Bentley, Stafford, UK
//   April 2021
//
//   This example and code is in the public domain and
//   may be used without restriction and without warranty.
//
```

```
#include <ez_SIPO8_lib.h>

#define max_SIPOs  1  // one 1 SIPO for this example
#define max_timers 0  // no timers required

// initiate the class for max SIPOs/timers required
SIPO8 my_SIPOs(max_SIPOs, max_timers);

int my_matrix7;  // used to keep the SIPO bank id

#define num_digits   16
uint8_t LSB_matrix_chars[num_digits] = {
  252, 96, 218, 242, 102, 182, 190, 224, 254, 246, 238, 62, 156, 122, 158, 142
};
#define LSB_DP_char 0b00000001  // "." value for LSBFIRST shiftout

uint8_t MSB_matrix_chars[num_digits] = {
  63, 6, 91, 79, 102, 109, 125, 7, 127, 111, 119, 124, 57, 94, 121, 113
};
#define MSB_DP_char 0b10000000  // "." value for MSBFIRST shiftout

void setup() {
  Serial.begin(9600);
  // create a bank of 1 SIPO using create_bank function:
  // data pin, clock pin, latch pin, number of SIPO this bank
  my_matrix7 = my_SIPOs.create_bank(8, 10, 9, 1); // data pin, clock pin, latch
pin, number of SIPO this bank
  if (my_matrix7 == create_bank_failure) {
    Serial.println(F("\nfailed to create bank"));
    Serial.flush();
    exit(0);
  }
  // print the bank data for confirmation/inspection
  my_SIPOs.print_SIPO_data();
}

void loop() {
  // keep running through the digits 0 to hex F, as defined by the
  // bit patterns in the preset array MSB_/LSB_matrix_chars
  do {
    // cycle 1 - MSBFIRST, with appended DP character "."
    my_SIPOs.set_bank_SIPO(my_matrix7, 0, 0b00000000);  // reset all LEDs in the
matrix to off/LOW
    my_SIPOs.xfer_bank(my_matrix7, MSBFIRST);
    delay(50);
    for (uint8_t digit = 0; digit < num_digits; digit++) {
      my_SIPOs.set_bank_SIPO(my_matrix7, 0, MSB_matrix_chars[digit] + MSB_DP_char);
// append "."
      my_SIPOs.xfer_bank(my_matrix7, MSBFIRST);
      delay(500);
    }
    // cycle 2 - LSBFIRST, no appended DP character
    my_SIPOs.set_bank_SIPO(my_matrix7, 0, 0b00000000);  // reset all LEDs in the
matrix to off/LOW
    my_SIPOs.xfer_bank(my_matrix7, LSBFIRST);
    delay(50);
    for (uint8_t digit = 0; digit < num_digits; digit++) {
      // if DP char "." required to be appended to a char, then add 'LSB_DP_char'
      // to the 'LSB_matrix_chars[digit]' parameter
      my_SIPOs.set_bank_SIPO(my_matrix7, 0, LSB_matrix_chars[digit]);
      my_SIPOs.xfer_bank(my_matrix7, LSBFIRST);
      delay(500);
    }
  } while (true);
}
```