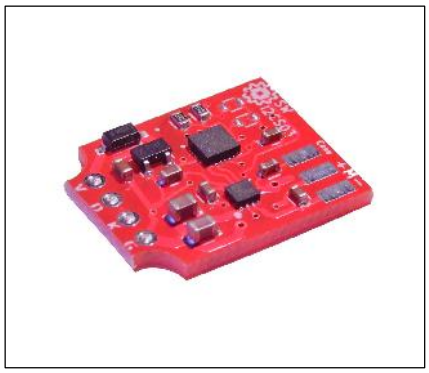
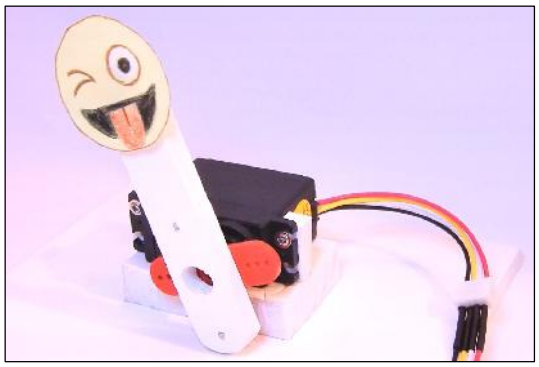
SWI2CS003
I2C Servo Board

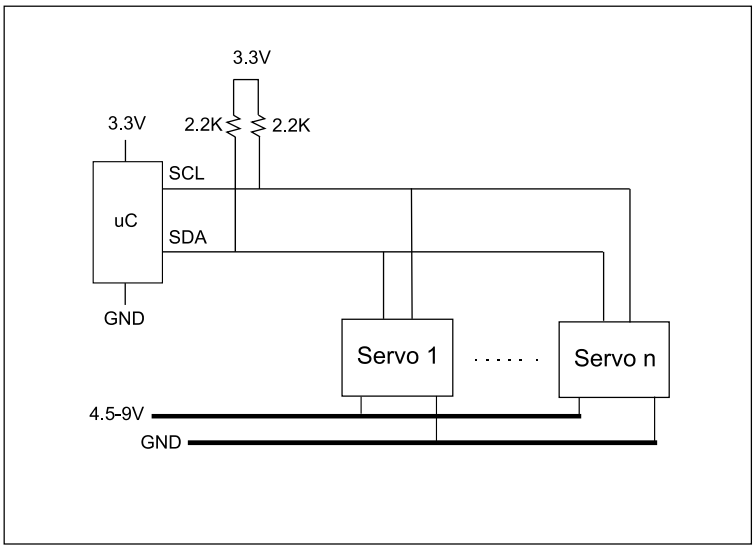




The SWi2CS series of boards allow easily converting a standard servo from the normal pulse-width control, to a smart but simple I2C interface. This allows for less wiring, pin-count, and controller overhead on your robotics project. Works with Arduino/RasPi and other boards that have an I2C port.

Features:
- Input voltage: 4.5 – 9V.
- Drive current: 1.5Amps.
- Fits standard 40mm, 3 to 6-Kg servo.
- Dimensions: 17x19mm
- Accessories: PH 4-pin connectors (24Ga wire)
- I2C interface (3.3v). Fast, smart yet simple and practical.
- Programmable I2C address. Have multiple servos on one bus.
- Control power, speed, ramp time and ramp profile.
- Read actual position, velocity, power.
- Programmable position sequence. Runs without controller.
- Optional CRC8 checksum on I2C read and write
- Example source and projects provided.

APPLICATION DIAGRAM

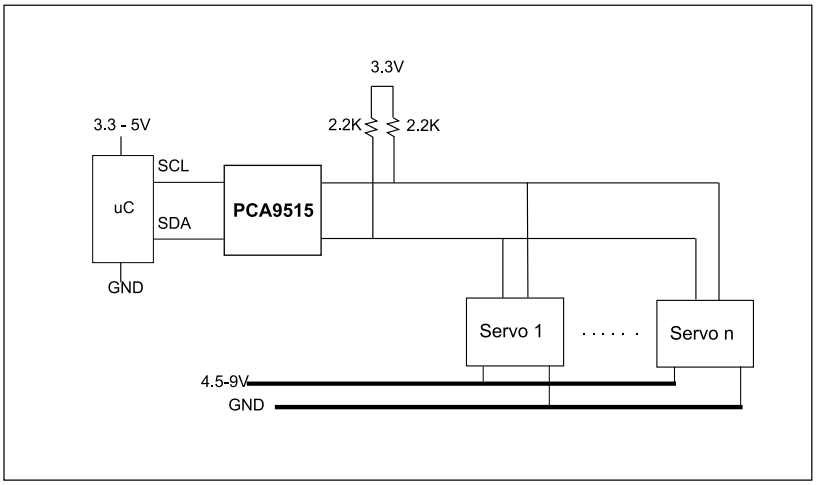


- The I2C pull ups to 3.3v are placed on the controller side. 2.2KOhm recommended.

- Make sure the main voltage supply and ground wires have enough thickness to carry the added system current with ease. 18-14Ga wire recommended.

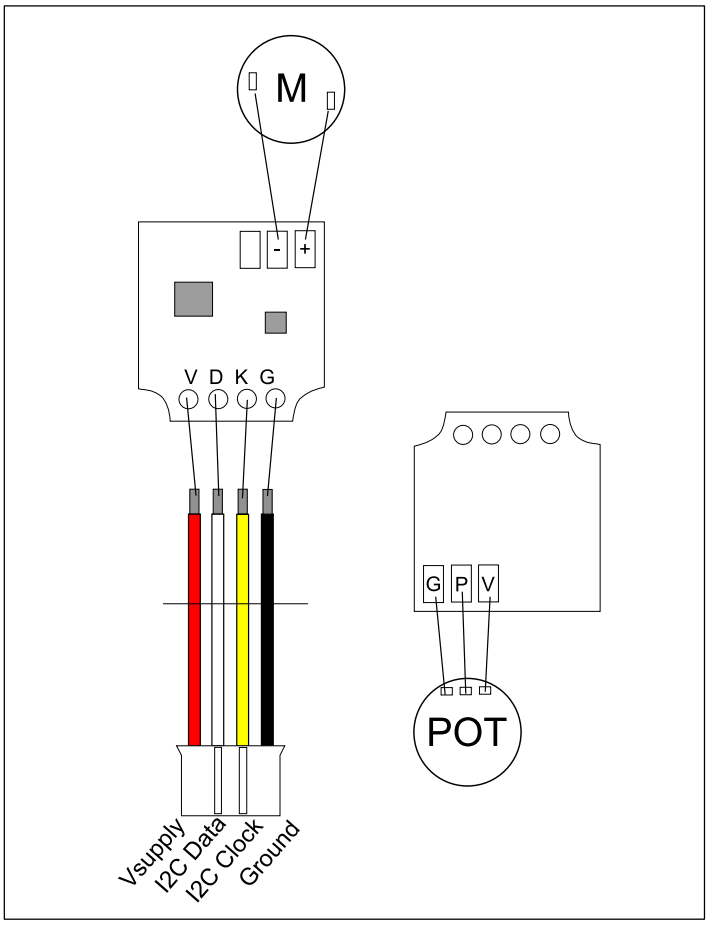
- Note that while the board can be supplied with up to 9v, servo motors are usually rated to 6-7v, and higher power is likely to reduce their service life.
  If running on 7-9v, you can try reducing the power setting via software (MAX_POWER register) to reduce this impact.

- If your controller runs on 5v, you can use a I2C driver module like the PCA9515B or similar to interface between the 5v and 3.3v domains:



- This driver is recommended regardless of the bus level, as it isolates the circuits and helps with speed and signal integrity.

BOARD INSTALLATION



This diagram illustrates the pin-out of the board where soldering to the servo takes place.
Incoming connector signals are:

Red : Vin 4.5-9V
White: I2C Data
Yellow: I2C Clock
Black : Ground

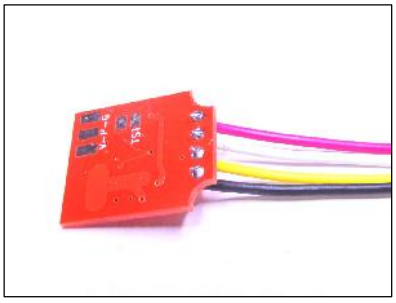
The two pads on the top right are for the motor – and + connections.
The three pads on the reverse are for the position sensing potentiometer wires.

Step 1:

- Clamp the servo from the motor side so it is easier to manipulate.
- Remove the servo's backside screws and remove the cover.
- Using your soldering iron, carefully de-solder the connections of the original servo board and remove it.
  You may have to wedge a small screwdriver under the board to slowly pry it off as you melt the solder, side to side.

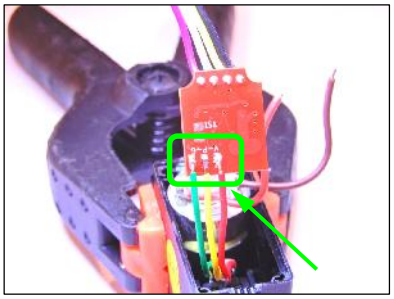
Step 2:

- Solder the provided female connector to the SWI2CS servo board. Be mindful of the color order.
- Solder joints should be round and shiny, remember that solder flux is your friend ;)



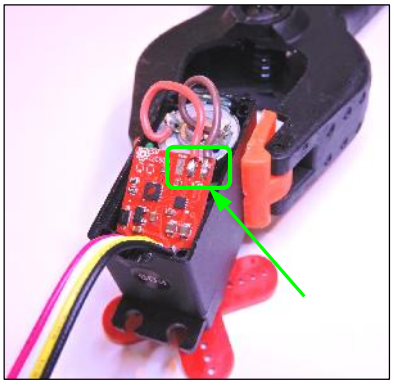
Step 3:

- Flip the board upside down and solder the pot wires to the pads labeled V-P-G, as shown.
- Use the provided wires if the pot has pins and no wires of its own. Put some tape over the pot contacts to avoid any shorts.
- If the pot pins are oriented not near the motor but rather towards the plastic case, then invert the wire order.



Step 4:

- Turn the board upright again and nicely accommodate the pot wires underneath, forming an 'S' pattern. Try to make the board lay flat or a little lower within the case space.
- Solder the motor wires to the pads labeled –M+ as shown. Depending on the servo type, the motor wire orientation may Need to be inverted, but you can do this via software as well.
- Accommodate the wires around so you can replace the servo cover without obstruction. Cut a little bit of material from the plastic cover notch to make more space for the four connector wires coming to the outside.



At this point you can screw the cover back in place, or you can first test the servo with power and I2C communication, to ensure everything works correctly.

IMPORTANT TIPS

- If the potentiometer leads are towards the plastic case (pot body is turned 180deg) the pot wiring order must be inverted.

- Depending on the gear configuration of a particular servo, the motor rotation can be either clockwise or counterclockwise.
You can start with a low power setting to test motion. If the servo runs away from the target position and goes to one of the ends, reverse the motor wire polarity, or change the motor polarity bit in the MOTOR_POLARITY register .

-Remember that the end ranges 0-20 and 1000-1024 are small keep out zones.
This is a safety feature so the servo can identify potentiometer failure or disconnect.
The servo won't move when at the very ends of travel.
Please move the servo manually from full lock position.

-Don't use continuous-rotation mode on a single-turn servo as you will crash on the ends and possibly suffer mechanical damage.
Default mode is CONTINUOUS_ROTATION=0 (Off)

-The default I2C address is 0x00.
If you have changed this and don't remember the current value, you can use one of the example code snippets, it loops through the address range and prints which board number responds.

I2C COMMUNICATION

The board uses the standard TWI (I2C compatible) interface bus, with a Data and a Clock signal lines (pull up resistors are required on the main controller side). It can work with a frequency of up to 200Khz, depending on bus length.

Write format:

| (Start) ADDRESS (W) | REGISTER | DATA_1 | DATA_N (Stop) |

Read format (compound read):

| (Start) ADDRESS (W) | REGISTER | (Start) ADDRESS (R) | DATA_1 | DATA_N (Stop) |

Note that you can access more than one register on the register space starting at the 'REGISTER' position, with each new 'DATA' byte automatically incrementing the register pointer.
If you write multiple registers on the same access transaction, any effect will happen only after the whole write is complete.
For register writing, please give the servo 1mS to digest the data before issuing another transaction. You can access another servo or carry another application task in the mean time.

Example code

These protocol details are only covered briefly in this document. There are example projects and wrapper routines available so you don't have to reinvent the wheel. Please visit the github repository:

www.Github.statorworks.i2cservo_examples\

Address assignment:

The board's default I2C address is 0x00. You can start testing with this default but can change it to any number in the range 0x04 - 0x70 by writing to register I2C_ADDRESS.
When changing address, make sure the board is the only one on the bus or the only one that has the current address. Tip: Place a small sticker dot and write the address on the servo.

The new address takes effect immediately and subsequent accesses should be made to the new address. The new address is recorded to on-board eeprom and later loaded on power up.

Optional CRC8 Chekcsum:

The I2C protocol works rather well in small buses up to 6 feet in length, and up to 200Khz. If you want to add even further robustness to your system you can use the available CRC8 checksum format on the access transactions.

The format differs to normal I2C packets in the following way:

- A '1' on the MSb of the 'REGISTER' byte indicates the packet is implementing CRC8.
- A 'LENGTH_N' byte is included after the 'REGISTER' byte. This is the count of DATA_N bytes.
- CRC8 (SMbus type 0x7 Polynomial) is calculated for all bytes including
  ADDRESS, REGISTER, LENGTH_N, DATA0, DATA_N, and appended as the last byte.

-A write packet would take the form:
| (Start) ADDRESS (W) | (1)REGISTER | LENGTH_N | DATA_1 | DATA_N | CRC8 (stop) |

In this case if the checksum fails due to a transmission error, the servo will ignore the write in its entirety. The application can later read the LAST_CRC8 register and compare it to the local calculation to see if last write was successful.

-A read packet would take the form:
(Start) ADDRESS (W) | (1)REGISTER | LENGTH_N | (Start) ADDRESS (R) | DATA_1 | DATA_N | CRC8(stop)|

Here the application calculates the checksum from the received packet and compares it to CRC8. If the checksum fails, the application can retry the reading transaction.

Again, there is example code and functions so you don't have to worry about these details.

REGISTER SPACE

OFFSET	NAME	R/W	RANGE	DEFAULT	DESCRIPTION
0x00	FIRMWARE_VERSION	R	1-N	1	Current software revision
0x01	I2C_ADDRESS	R/W	0-127	0	Bus address. Written new address is saved immediately Further access must be to the new address
0x02	STOP	W	1	0	Stop Immediately with no ramping, and enter park state
0x03	PARK_TYPE	R/W	0-255	1	0 = Coast, free motor. No power consumption. 1 = Brake, driver ties up motor leads. No power consumption. 2 = Active hold with compliance with power = MAX_POWER 3-100 = Active hold with compliance. Power: From this number
0x04	MOTOR_POLARITY	R/W	0-1	1	Used as correction if wires are soldered backwards to motor 0=CCW 1=CW
0x05	CONTINUOUS ROTATION	R/W	0-1	0	0 = single turn, normal servo operation. 1 = Continuous rotation with MAX_POWER as speed control
0x06	MAX_POWER	R/W	0-100	80	0 % = no power 100% = Maximum power when seeking or holding position
0x07	MAX_SPEED_H	R/W	0-5000	2000	Ramp profile speed limit. Actual speed reached will depend on servo type and load Typically 2000 units per second for 0.16s-60deg servo
0x08	MAX_SPEED_L	R/W			
0x09	RAMP_TIME_H	R/W	1-10000	250	mSeconds to reach MAX_SPEED (if power and load permit)
0x0A	RAMP_TIME_L				
0x0B	RAMP_CURVE	R/W	0-100	50	0 = Linear ramp profile 100 = Sine ramp profile
0x0C	DEADBAND WINDOW	R/W	0-20	1	Target position tolerance. Helps with flexibility.
0x0D	TARGET_POSITION_H	R/W	0-1024	0	Write desired position, this will override any current motion.
0x0E	TARGET_POSITION_L	R/W			
0x0F	TARGET_POSITION_NEXT	R/W	0-1024	0	Push new position here without having to wait for current motion completion.
0x10	TARGET_POSITION_NEXT	R/W			
0x11	CURRENT_STATE	R	0-2	0	0:Parked 1:seeking 2:Running stored program
0x12	CURRENT_POSITION_H	R	0-1024	0	Current position
0x13	CURRENT_POSITION_L	R			
0x14	CURRENT_VELOCITY_H	R	- 5000 +5000	0	Current velocity in Position units/Sec
0x15	CURRENT_VELOCITY_L	R			
0x16	CURRENT_POWER	R	0-100	0	Current power % being applied to exert motion or hold
0x17	CURRENT_TEMPERATURE	R	0-100	25	CPU temperature in Celsius
0x18	LAST_CRC8	R	0-255	0	When using i2c crc scheme. See example projects.
0x19	PROGRAM_POSITION_N_H	R/W	0-1024	0	(N) x8 slots.. .. 0=unused slot/end of sequence
0x1A	PROGRAM_POSITION_N_L	R/W			
0x29	PROGRAM_REPS	R/W	0-255	0	Reps to do programmed sequence upon power up. 255=inf
0x2A	SAVE	W	1	0	Saves all registers. Motion settings loaded on next power up.

Registers - Detailed Description

0x00 FIRMWARE_VERSION RW: RW RANGE: 0-1 DEFAULT: 1

If there is ever a firmware revision significantly affecting behavior, this number can be checked by the application.
You can read this register to confirm communication.

0x01 I2C_ADDRESS RW: RW RANGE: 0-1 DEFAULT: 0

Written address is saved immediately and further access must be to new address.
You can read this register to confirm communication.
You can iterate through the address range 0x00 to 0x70 to see which device(s) respond.

0x02 STOP RW: W RANGE: 0-1 DEFAULT: 0

1 =
Stop motion immediately with no ramping, and enter park state as per PARK_TYPE.

0x03 PARK_TYPE RW: RW RANGE: 0-1 DEFAULT: 0

On power up, or after reaching desired target position, the servo will enter one of The following position hold modes:

0 = Coast, free motor. No power consumption.
External force can easily move servo.

1 = Brake, driver ties up motor circuit. No power consumption.
External force can move servo but there is some considerable resistance.

2 = Active hold with compliance.
External force can move the servo but resistance gradually increases.
Resistance power is based on MAX_POWER register value.

3-100 = Active hold with compliance.
External force can move the servo but resistance gradually increases.
Same as option 2 but power % is based on this 3-100 value.
This allows you to set a hold force different than the one used for motion.

0x04 MOTOR_POLARITY RW: RW RANGE: 0-1 DEFAULT: 0

Invert motor polarity. Used if motor wires are soldered backwards to board.
For consistency, It is advised you correct the soldering instead, and use this option only for testing.

0x05 CONTINUOUS ROTATION RW: RW RANGE: 0-1 DEFAULT: 0

The default servo behavior is single turn position-seek. But servos that have been modified for continuous rotation can be controlled as well.
In that case the MAX_POWER register will control the power and polarity applied to the motor.
Don't use this option with regular servos as you can crash the gearbox or pot.

0x06 MAX_POWEER RW: RW RANGE: 0-100 DEFAULT: 80

Control the maximum power % applied when seeking the target position.
This is also the power to hold the target position once reached (see PARK_TYPE).

When in 'continuous rotation' mode, the range will be -100 to 100% to control both power and direction.

0x07, 0x08 MAX_SPEED_H-L RW: RW RANGE: 0-5000 DEFAULT: 2000

High and low bytes for 16-bit MAX_SPEED value.
The servo will cap the maximum speed when seeking the target position.
Units are in position-units per second.

0x09, 0x0A RAMP_TIME_H-L RW: RW RANGE: 1-10000 DEFAULT: 250

High and low bytes for the 16-bit RAMP_TIME value.
This controls the time it takes to accelerate from zero to MAX_SPEED (or MAX_SPEED to zero) in mS units. This is the same as the time from MAX_SPEED to zero, when seeking the target position.

For example a setting of 1000 (H=0x03, L=0xE8) will reach MAX_SPEED in 1 second.
Note that MAX_SPEED may not necessarily be reached, depending on the distance to be traveled, power setting, and mechanical load.

0x0B RAMP_CURVE RW: RW RANGE: 0-100 DEFAULT: 50

This controls the servo motion profile of the acceleration ramp.
A linear profile provides constant acceleration to/from MAX_SPEED, while a sinusoidal 'second order' profile provides lower acceleration around zero and MAX_SPEED, giving a smoother motion.

0 = Linear ramp
100 = Sine ramp

You can adjust this setting to fit the mechanical load to get a more organic motion. Please note that higher values slightly alter the actual total ramp-time.

0x0C DEADBAND_WINDOW RW: RW RANGE: 0-20 DEFAULT: 1

This adjusts the target position tolerance window.
When seeking the target position, the position will be considered achieved when within N position units either side of the target.

You can open this window to higher values on servos that lack mechanical accuracy, or if the load comes as an issue to actually reaching the target.

It is also useful with the programmable position sequence (PROGRAM_POSITION _N) and the TARGET_POSITION_NEXT features. You trade of some accuracy for flexibility, so that the sequenced motions are not delayed.

0x0D 0x0E

TARGET_POSITION_H-L RW: RW RANGE: 0-1024 DEFAULT: 0

High and low bytes for the 16-bit TARGET_POSITION value.

Write this register pair to tell the servo where to go.
Any current motion will be overridden by this and the servo will ramp as per MAX_POWER, RAMP_TIME, RAMP_CURVE settings to seek the desired position.

Please note that the end ranges 0-20 and 1000-1024 are reserved. This is a safety feature so the servo can identify potentiometer failure or disconnection.
Servo won't move if currently at one of the ends.
Also position setting will be truncated to the 20-1000 range.

0x0F 0x10
TARGET_POSITION_NEXT_H-L RW: RW RANGE: 0-1024 DEFAULT: 0

High and low bytes for the 16-bit TARGET_POSITION_NEXT value.

You can push new target position values here and they will be executed in order as each of the positions is achieved.
This way if you want a few motions back to back you don't have to wait or poll the servo for completion.

You can push up to eight(8) positions on this internal stack. As each move is completed, the top of the stack is freed up and you can keep pushing further values.
Values added when the stack is already full are discarded.

Writing any value to the TARGET_POSITION register or issuing a STOP write will cancel the current motion and clear this stack.

Tip: Depending on the load, you can open the DEADBAND_WINDOW value to ensure prompt completion of the sequence.

Please note that position end ranges 0-20 and 1000-1024 are reserved. This is a safety feature so the servo can identify potentiometer failure or disconnection.
Servo won't move if currently at one of the ends.
Also position setting will be truncated to the 20-1000 range.

0x11 CURRENT_STATE RW: R RANGE: 0-3 DEFAULT: 0

0 = Parked on target position
1 = Seeking target position
2 = Running programmable sequence
3 = Continuous rotation mode

0x12 0x13
CURRENT_POSITION_H-L RW: R RANGE: 0-1024 DEFAULT: 0

High and low bytes for the 16-bit CURRENT_POSITION value.

Read the present position here regardless of operating state.

0x14 0x15
CURRENT_VELOCITY_H-L **RW: R** **RANGE: -5000 to 5000** **DEFAULT: 0**

High and low bytes for the 16-bit CURRENT_VELOCITY value.

Read the present velocity here regardless of operating state.
Units are position units per second.

0x16 CURRENT_POWER **RW: R** **RANGE: -100 to 100** **DEFAULT: 0**
Read the present power % being applied.
A negative value indicates power is counter-clockwise, positive is clockwise.

0x17 CURRENT_TEMPERATURE **RW: R** **RANGE: 0-100** **DEFAULT: 0**
Read the current CPU temperature here in degrees centigrade.
This can give an indication of overall servo temperature, normal function is 30-40*

0x18 LAST_CRC8 **RW: R** **RANGE: 0-255** **DEFAULT: 0**
When using the optional I2C CRC8 feature, the servo stores the crc8 value for the last register write transaction.
The application can read this value to confirm write and crc were successful.
Note that this read itself can be performed with the CRC8 feature as well.

0x19 0x1A + [Nx2]
PROGRAM_POSITION_H-L [N] **RW: RW** **RANGE: 0-1024** **DEFAULT: 0**

High and low bytes for the 16-bit PROGRAM_POSITION value.
You can store up to eight(8) positions to automatically run in sequence on power up, without a need for the main controller.

Each 16-bit position slot has a High and Low byte register:
P0: 0x19, 0x1A
P1: 0x1B, 0x1C
P2: 0x1D, 0x1E
P3: 0x1F, 0x20
P4: 0x21, 0x22
P5: 0x23, 0x24
P6: 0x25, 0x26
P7: 0x27, 0x28

- A zero(0) on any given slot marks the end of the sequence

  For example, with an assignment of the form

  U16 Position_Array[8] = { 100, 400, 900, 0, 0, 0, 0, 0 };

  The servo will move to position 100, then 400, then 900.
  PROGRAM_REPS will be decremented.
  If PROGRAM_REPS is non-zero the sequence is started again.

- To enable the sequence, set the PROGRAM_REPS register to a non-zero value.
  Then write the SAVE register with one(1).
  Next power up will run the motion sequence automatically.

- To disable the sequence, write the position values to zero(0), or the PROGRAM_REPS register to zero(0).
  Then write the SAVE register with one(1).

  Tip: Depending on the load, you can open the DEADBAND_WINDOW value to ensure each position is promptly hit without delaying the sequence. This is a tradeoff between accuracy and agility.

0x29 PROGRAM_REPS **RW: RW** **RANGE: 0-255** **DEFAULT: 0**

0 = Disabled/Stop
1-254 = Run N times
255 = Infinite

0x2A SAVE **RW: W** **RANGE: 0-1** **DEFAULT: 0**

1 = Save all registers to non-volatile memory

Upon power up, the run settings registers are loaded as current and used for motion until overwritten.

Upon power up, if there are position values on the PROGRAM_POSITION registers and PROGRAM_REPS is non-zero, the programmable sequence will start.

EXAMPLES

The following pseudo code shows some basic read and write cases.
For full example projects, please go to

github.statorworks/i2cServo/examples

Address assignment:

Raw:
```
Data[0] = NEW_ADDRESS;
I2c_write(ADDRESS, REG_I2C_ADDRESS, &Data[0], 1);
```

Using library:
```
I2CServo_SetAddress(ADDRESS, NEW_ADDRESS, 0);
```

Set Max Power:

Raw:
```
Data[0] = MAX_POWER;
I2c_write(ADDRESS, REG_I2C_SET_MAX_POWER, &Data[0], 1);
```

Using library:
```
I2CServo_SetMaxPower(ADDRESS, MAX_POWER, 0);
```

Set target position:

Raw:
```
Data[0] = POS>>8;
Data[1] = POS&0xFF;
I2c_write(ADDRESS, REG_I2C_TARGET_POSITION, &Data[0], 2);
```

Using library:
```
I2CServo_SetTargetPosition(ADDRESS, POS, 0);
```

Read target position:

Raw:

```
i2c_read(ADDRESS, REG_I2C_TARGET_POSITION, &Data[0], 2);
POS = Data[0]<<8 | Data[1];
```

Using library:

```
I2CServo_GetTargetPosition(ADDRESS, &POS, 0);
```

Set target position, with CRC8 checksum:

Raw:

```
REG    = REG_I2C_TARGET_POSITION | 0x80;   //msb=1: crc use
//
Buff[0] = ADDRESS;
Buff[1] = REG;
Buff[2] = 2; //data length
Buff[3] = POS>>8;
Buff[4] = POS&0xFF;
Buff[5] = calc_crc8(&Buff[0], 5);
//
I2c_write(ADDRESS, REG, &buff[2], 4);
```

Using library:

```
I2CServo_SetTargetPosition(ADDRESS, POS, 1);   //1:use crc8  scheme
```