

# BITWISE MANIPULATION IN A SCRIPT

## 1. Introduction

Senquip devices can run custom scripts, allowing extra flexibility. The scripting language is called mJS and is a restricted JavaScript engine.

Bitwise manipulation involves performing operations, including setting, clearing, toggling, and shifting bits within a binary representation of data.

This can be useful for interpreting status bits and setting control bits in Bluetooth, CAN, Modbus, and other messages.

This application note describes how to perform bit manipulation in scripts written in mJS for Senquip devices.

Further details on the Senquip scripting language can be found in the [Device Scripting Guide](#).

## 2. Bitwise Operations

A summary of bitwise operations supported in mJS and a refresher on the operation of basic logic gates and the bitwise functions that they perform is presented below.

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shifts left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off
>>>	Zero fill right shift	Shifts right by pushing zeros in from the left, and let the rightmost bits fall off

Be sure to use brackets to set the order of priority when using multiple operations in the same line.

### 2.1. NOT Gate

The output of a NOT gate is always the opposite of the input.

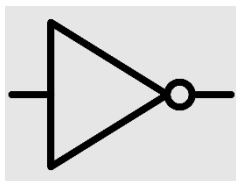


Figure 1 - NOT Gate Symbol

<b>A</b>	<b>~A</b>
0	1
1	0

## 2.2. AND Gate

For the output of an AND gate to be high, both the inputs need to be high.

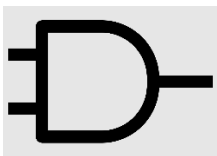


Figure 2 - OR Gate Symbol

<b>A</b>	<b>B</b>	<b>A &amp; B</b>
0	0	0
0	1	0
1	0	0
1	1	1

## 2.3. OR Gate

For the output of an OR gate to be high, either or both the inputs need to be high.

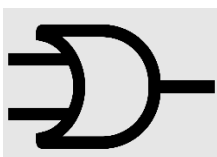
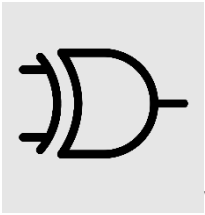


Figure 3 - OR Gate Symbol

<b>A</b>	<b>B</b>	<b>A &amp; B</b>
0	0	0
0	1	1
1	0	1
1	1	1

## 2.1. XOR Gate

For the output of an exclusive OR (XOR) gate to be high, only 1 input must be high, but not both.



word when I save to pdf I get

Figure 4 - XOR Gate Symbol

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

## 2.2. Shift Left

Shifts left by pushing zeros in from the right and let the leftmost bits fall off.

Example:

```
0b00000000000000000000000000000101 << 2 = 0b000000000000000000000000000010100
```

## 2.3. Shift Right

Shifts right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off.

Example:

```
0b00000000000000000000000000000101 >> 2 = 0b00000000000000000000000000000001
```

```
0b11111111111111111111111111111011 >> 2 = 0b11111111111111111111111111111110
```

## 2.4. Zero Fill Shift Right

Shifts right by pushing zeros in from the left, and let the rightmost bits fall off.

Example:

```
0b00000000000000000000000000000101 >> 2 = 0b00000000000000000000000000000001
```

```
0b00000000000000000000000000000101 >> 2 = 0b00111111111111111111111111111110
```

## 3. Data Types

All numbers in JavaScript are stored as 64-bit floating point. It is however important to understand the way numbers are stored as they impact how bitwise operations are performed.

**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
4 of 17

### 3.1. Unsigned Numbers

Unsigned numbers are always assumed to be positive, and all the bits can be used to store the magnitude of the number. An 8-bit unsigned number can store a value in the range 0 to 255.

Some examples are given in the table below:

Number	Bit 7	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	1	0	1
255	1	1	1	1	1	1	1	1

### 3.2. Signed Numbers

Signed numbers are stored in two's complement format which is the standard for all computer systems.

In the two's complement representation, the leftmost bit indicates the sign. If the leftmost bit is 0, the number is interpreted as a positive number and if the leftmost bit is a 1, then the number is negative. An 8-bit signed number can store a value in the range -128 to 127.

Number	Sign	Bit 6	Bit 5	Bit 4	Bit 3	Bit 2	Bit 1	Bit 0
5	0	0	0	0	0	1	0	1
-5	1	1	1	1	1	0	1	1
-128	1	0	0	0	0	0	0	0
127	0	1	1	1	1	1	1	1

To obtain the absolute value of the negative number, all the bits are inverted then 1 is added to the result. The result is then read as an unsigned number. An example is given below.

negative

-5		1	1	1	1	1	0	1	1
	Invert bits	0	0	0	0	0	1	0	0
5	Add 1	0	0	0	0	0	1	0	1



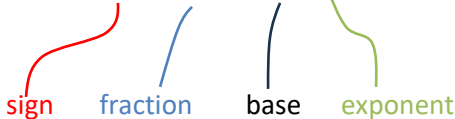
When dealing with signed integers, not considering the sign bit during bitwise operations can lead to unexpected results. Be aware of the two's complement representation and its implications on signed integers.

<b>Document Number</b> APN0027	<b>Revision</b> 1.1	<b>Prepared By</b> NGB	<b>Approved By</b> NB
<b>Title</b> Bit Manipulation in a Script			<b>Page</b> 5 of 17

### 3.3. Floating Point Numbers

Floating point numbers are numbers that may contain fractions. A standard, IEEE 754, describes the way floating numbers are stored. IEEE 754 uses scientific notation to store numbers.

$$42000 \rightarrow 1 \times 4.2 \times 10^5$$

$$-0.0042 \rightarrow -1 \times 4.2 \times 10^{-3}$$


The diagram shows the components of the scientific notation examples above. Colored lines connect the terms to their respective parts: a red line from 'sign' to '-1', a blue line from 'fraction' to '4.2', a black line from 'base' to '10', and a green line from 'exponent' to '5' and '-3'.

As can be seen from the example above, the number representation can be split into three parts:

- **sign**
- **fraction — the valuable digits (the meaning, the payload) of the number**
- **exponent — controls how far and in which direction to move the decimal point in the fraction**

The base can be omitted as it is always 2 (binary).

Instead of using all 16 bits (or 32 bits, or 64 bits) to store the fraction of the number, we share the bits and store a sign, exponent, and fraction at the same time. Depending on the number of bits that we're going to use to store the number we end up with the following splits:

Floating Point Format	Total Bits	Sign Bits	Exponent Bits	Fraction Bits	Base
16	16	1	5	10	2
32	32	1	8	23	2
64	64	1	11	52	2

Since we now know the sign, fraction, base, and exponent, we can regenerate the floating-point number.

A thorough explanation of floating-point numbers can be found [here](#).

Bitwise operations do not make sense for floating point numbers. MJS will convert floating point numbers to 32-bit unsigned integers before performing bitwise operations.

## 4. Data Types in mJS

All numbers in JavaScript are treated as 64-bit floating-point numbers. Senquip does however define certain data types that are used by functions when interpreting numbers.

For instance, given the following 8-bit number:

Hex	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
F5	1	1	1	1	0	1	0	1

To interpret the number, we need to know if it is signed or unsigned. If it is a signed number, the value will be -11; if it is unsigned, the value will be 245.

<b>Document Number</b> APN0027	<b>Revision</b> 1.1	<b>Prepared By</b> NGB	<b>Approved By</b> NB
<b>Title</b> Bit Manipulation in a Script			<b>Page</b> 6 of 17

The table below describes the data types defined in the Senquip scripting language and the range of numbers supported.

Data Type	Description	Minimum Value	Maximum Value
SQ.U8	Unsigned 8 bit	0	255
SQ.S8	Signed 8 bit	-128	127
SQ.U16	Unsigned 16 bit	0	65,535
SQ.S16	Signed 16 bit	-32,768	32,767
SQ.U32	Unsigned 32 bit	0	4,294,967,295
SQ.S32	Signed 32 bit	-2,147,483,648	2,147,483,647
SQ.U64	Unsigned 64 bit	0	18,446,744,073,709,551,615
SQ.S64	Signed 64 bit	-9,223,372,036,854,775,808	9,223,372,036,854,775,807
SQ.FLOAT	4-byte floating point (32 bit)		
SQ.DOUBLE	8-byte floating point (64 bit)		

**NOTE:**

Bitwise operators in mJS always operate on 32-bit operands. Internally, 64-bit floating point are converted into 32-bit unsigned integers before performing bitwise operations. They are then converted back to 64-bit numbers to return the result.

That mJS performs bitwise operations on 32-bit unsigned integers can be demonstrated using a simple script that declares a number equal to the maximum allowed in a 32-bit unsigned integer and a number 1 greater than this. It then performs an AND function with the 2 numbers and all 1's. The result should be the initial number. As can be seen in the result however, the bitwise function works for the 32-bit number but not the 64-bit number.

```
load('senquip.js');

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  let num32 = 4294967295; //max number allowed in a 32-bit unsigned int
  let num64 = 4294967296; // one larger than that allowed in a 32-bit unsigned int

  SQ.dispatch(1,num32 & 0xffffffff);
  SQ.dispatch(2,num64 & 0xffffffffffffffff);
}, null);
```

**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
7 of 17

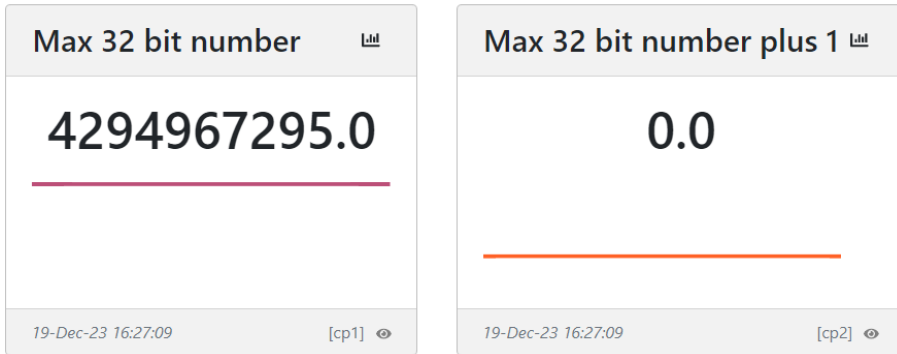


Figure 5 - Bitwise Operations Are Limited to 32 Bits

Do not try to perform bitwise operations on numbers larger than 32 bits.

## 5. Testing a Bit

Testing a bit can be useful where a single bit is used as a status indicator, for instance in a Kensho Controller with a Modbus interface, register 40022 tells us the current state of the controller. We need to test various bits to determine the state of the controller.

40022	-----	Auto/Manual mode	Bit 0 1 = Auto 0 = Manual Bit 1 1 = Engine Running Bit 2 1 = Warm Up Bit 3 1 = Line Fill Bit 4 1 = Cooldown Bit 5 1 = Common Alarm
-------	-------	------------------	--

Figure 6 - Kensho Auto/Manual Mode Register

To test the status of an individual bit, we need to isolate that bit. We can achieve this by forcing all bits to zero, except the one we want to test by using a mask and a bitwise AND. The mask below contains a 1 only in bit position 5 and so when a bitwise AND is performed with this mask, all bits will be cleared except the bit in position 5.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
<b>Mask</b>	0	0	1	0	0	0	0	0

In our example with the Kensho Controller, let's assume that we want to test if the engine is running. We therefore need to test to see if bit 1 is set. Let's say we have read the register, and the following is returned.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
<b>Register 40022</b>	0	0	0	0	0	1	1	1
<b>Mask</b>	0	0	0	0	0	0	1	0
<b>40022 AND Mask</b>	0	0	0	0	0	0	1	0

If the result is non-zero, then the bit is set, and the engine is running.

In mJS, it is easier to work with Hex masks rather than binary masks. Using the same example, but showing the register and mask in hex, the example becomes:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0		Hex
<b>Register 40022</b>	0	0	0	0	0	1	1	1		0x07
<b>Mask</b>	0	0	0	0	0	0	1	0		0x02
<b>40022 AND Mask</b>	0	0	0	0	0	0	1	0		0x02

A script for a Senquip device that tests bit 2 in a Modbus register and displays a message on the Senquip Portal indicating when the machine is running, is shown below:

```
load('senquip.js');

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  let reg = obj.mod1; // mod1 is configured to read register 40022
  let mask = 0x02;

  if((reg & mask) !== 0){
    SQ.dispatch(1, "Running");
  }
}, null);
```

## 6. Setting a Bit

Setting a single bit is useful in control applications for instance to activate an alarm or enable a specific function.

To set an individual bit and leave all others in their current state, we can again use a mask, but this time, we will use a bitwise OR. The mask below contains a 1 only in bit position 2 and so when a bitwise OR is performed with this mask, all bits will be left alone, except the bit in position 2 which will be set.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
<b>Mask</b>	0	0	0	0	0	1	0	0

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0		Hex
<b>Register</b>	1	0	0	0	0	0	1	0		0x82
<b>Mask</b>	0	0	0	0	0	1	0	0		0x04
<b>Register OR Mask</b>	1	0	0	0	0	1	1	0		0x86



<b>Document Number</b> APN0027	<b>Revision</b> 1.1	<b>Prepared By</b> NGB	<b>Approved By</b> NB
<b>Title</b> Bit Manipulation in a Script			<b>Page</b> 9 of 17

Note that we can set multiple bits at the same time by including additional 1's in our bit mask. In the example in the table below, bits 2 and 0 are both set at the same time.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0		Hex
<b>Register</b>	1	0	0	0	0	0	1	0		0x82
<b>Mask</b>	0	0	0	0	0	1	0	1		0x05
<b>Register OR Mask</b>	1	0	0	0	0	1	1	1		0x87

## 7. Clearing a Bit

Clearing a single bit performs the opposite function to setting a bit.

To clear an individual bit and leave all others in their current state, we use a bitwise AND where all bits are set to 1 except for the bit we want to clear, which will be set to zero.

In the example below, bit 2 will be cleared.

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0		Hex
<b>Register</b>	1	0	0	0	0	1	1	1		0x87
<b>Mask</b>	1	1	1	1	1	0	1	1		0xfb
<b>Register OR Mask</b>	1	0	0	0	0	0	1	1		0x83

A simple script that clears bit 2 is shown below:

```
load('senquip.js');

let reg = 0x87;

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  let mask = 0xfb;
  reg = reg & mask;

  SQ.dispatch(1, reg);
});
```

Note that we can clear multiple bits at the same time by including additional 0's in our bit mask.

## 8. Toggling a Bit

Toggling a bit can be advantageous in situations where you want to alternate between two states without explicitly checking the current state.

To toggle an individual bit and leave all others in their current state, we can again use a mask, but this time, we will use a bitwise XOR. The mask below contains a 1 only in bit position 2 and so when a bitwise XOR is performed with this mask, all bits will be left alone, except the bit in position 2 which will be toggled.

**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
10 of 17

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0
<b>Mask</b>	0	0	0	0	0	1	0	0

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
<b>Register</b>	1	0	0	0	0	0	1	1	0x83
<b>Mask</b>	0	0	0	0	0	1	0	0	0x04
<b>Register XOR Mask</b>	1	0	0	0	0	1	1	1	0x87

Applying the same mask again toggles the bit:

	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1	bit 0	Hex
<b>Register</b>	1	0	0	0	0	1	1	1	0x87
<b>Mask</b>	0	0	0	0	0	1	0	0	0x04
<b>Register XOR Mask</b>	1	0	0	0	0	0	1	1	0x83

The script below toggles bit 2 after each measurement cycle when the data handler runs.

```
load('senquip.js');

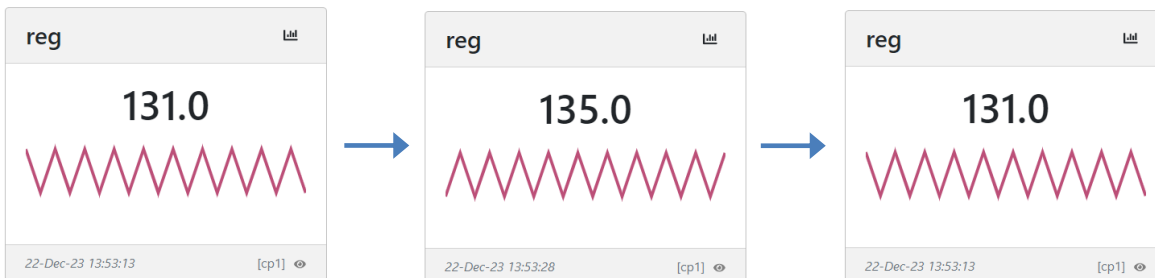
let reg = 0x83;

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  let mask = 0x04;
  reg = reg ^ mask;

  SQ.dispatch(1, reg);
}, null);
```

As expected, the output toggles between 0x83 (131) and 0x87 (135).



**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
11 of 17

## 9. Extracting Multiple Bytes from a String

CAN and Bluetooth messages are formatted as an ID and data – Bluetooth also reports RSSI. The data portion of the message is a string made up of multiple bytes. Senquip provides a [Parse](#) function to facilitate extraction of bytes from a string.

For example, a Senquip device with BLE enabled delivers the following message from an ELA Bluetooth beacon that is measuring temperature.

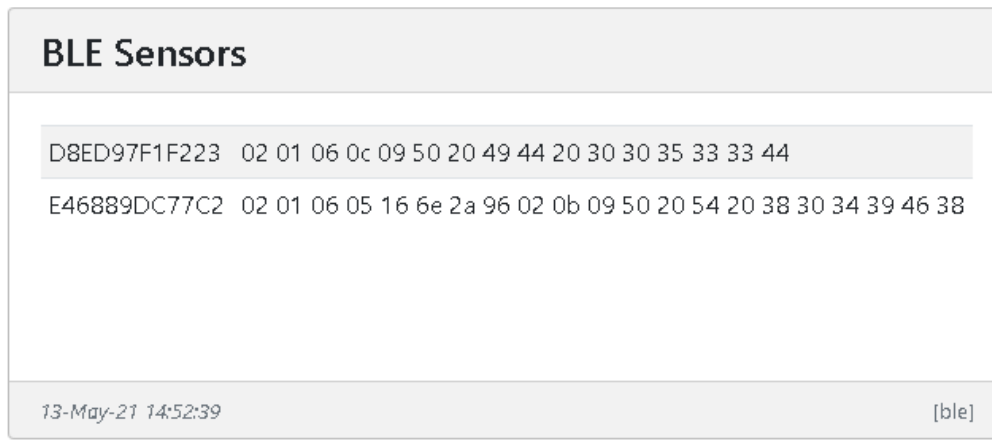


Figure 7 - Typical Bluetooth Hex Message

The data portion of the BLE message is shown below, with the temperature represented by “7d 02”.

“02 01 06 05 16 6e 2a 7d 02 0b 09 50 20 54 20 38 30 34 39 46 38”

To extract portions of the message, we use the Parse function. The Parse function convert a string to a number using the base given. Most messages will be strings in hex format. Zooming in on the first section of the message that contains the temperature, we have:

Position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Character	0	2	0	1	0	6	0	5	1	6	6	e	2	a	7	d	0	2

To extract the temperature, we use the *Parse* function.

SQ.parse(string, start, length, base, type)

- string: The string to parse.
- start: The starting character index. (0 = the first character in the string).
- length: The number of characters in the string to parse from the start position.
- base: The numerical base used when parsing the string. (10 = Decimal, 16 = Hexadecimal). A special case applies when when base is (-16): the byte order will be reversed.

<b>Document Number</b> APN0027	<b>Revision</b> 1.1	<b>Prepared By</b> NGB	<b>Approved By</b> NB
<b>Title</b> Bit Manipulation in a Script			<b>Page</b> 12 of 17

- type: (Optional) The number type used to interpret the parsed data. This parameter should be one of the encoding type constants. Default: SQ.S64

In this case, the temperature starts at position 14, the number of characters is 4, the base is 16 (hex). We use base -16 as the datasheet specifies that the bytes are in reverse order. The 16 bits that we end up with are stored as signed data and so we use the SQ.S16 type. The datasheet also states that the temperature needs to be scaled by 0.01.

Applying the Parse function, we get: 0x027d = 637.0, or a scaled temperature of 6.37 deg C.

## 10.CAN Bus Example

In 9, we looked at how to extract bytes from a string. We will now look at a CAN example where we need to extract bytes from a string and then separate out bits to extract status information.

The J1939 Tire Condition message is shown in Figure 8.

<b>pgn65268 - Tire Condition - TIRE -</b>			
Transmission Repetition Rate:		10 s	
Data Length:		8 bytes	
Data Page:		0	
PDU Format:		254	
PDU Specific:		244	
Default Priority:		6	
Parameter Group Number:		65268 ( 00FEF4 <sub>16</sub> )	
Bit Start Position /Bytes	Length	SPN Description	SPN
1	8 bits	Tire Location	929
2	1 byte	Tire Pressure	241
3-4	2 bytes	Tire Temperature	242
5.1	2 bits	CTI Wheel Sensor Status	1699
5.3	2 bits	CTI Tire Status	1698
5.5	2 bits	CTI Wheel End Electrical Fault	1697
6-7	2 bytes	Tire Air Leakage Rate	2586
8.6	3 bits	Tire Pressure Threshold Detection	2587

*Note: In the original image, green annotations highlight 'byte 5, bit 3' for bit 5.1 and '2 bits' for bit 5.3. A green arrow points from 'SPN 1698' to bit 5.3.*

**Tire Condition Message NOTE:** Message has to repeated as necessary to transmit all available information. This method of location identification requires individual SPNs to be assigned to report failures specific to each individual component (I.e. each tire, each axle, etc.).

Figure 8 - J1939 Tire Condition Message

We will write a script to extract the tire status. Tire status is specified by SPN 1669 and is contained in byte 5, bits 3 and 4. Figure 9 gives a description of the Tire Status SPN.

**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
13 of 17

<b>spn1698 - CTI Tire Status</b> - Indicates the status of the tire.	
00	Ok (no fault)
01	Tire leak detected
10	Error
11	Not Supported
Bit Length:	2 bits
Type:	Measured
Suspect Parameter Number:	1698
Parameter Group Number:	[ 65268 ]

Figure 9 - SPN 1698 Description

A Tire Condition CAN message was collected and is shown in Figure 10.

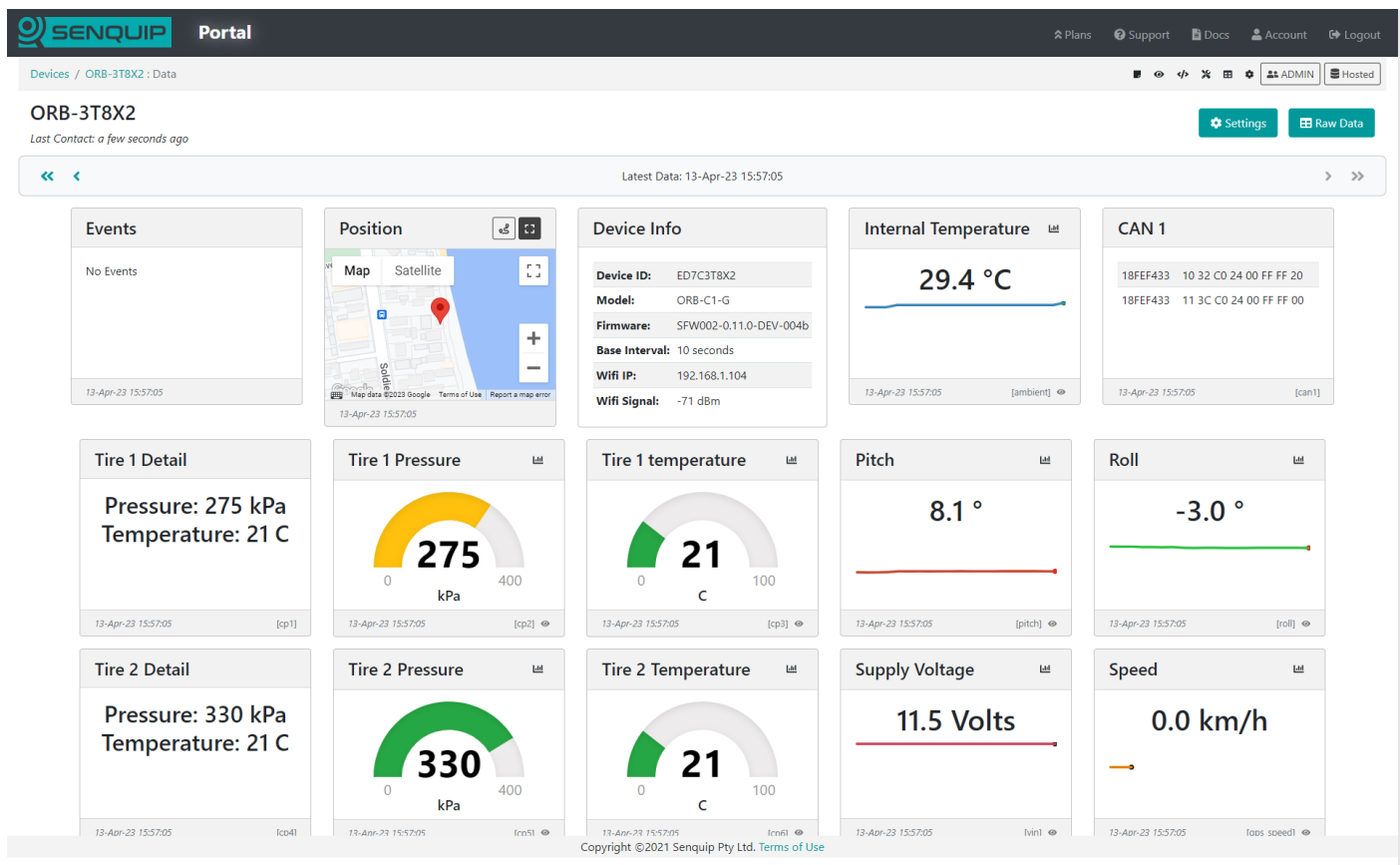


Figure 10 - Tyre Pressure Monitoring

The message data is as follows with byte 5 highlighted. J1939 specifies that the first data byte is sent first and is referenced as byte 1. The least significant bit of the data byte are on the right and are referenced as bit 1.

**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
14 of 17

<b>Position</b>	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
<b>Character</b>	1	0	3	2	c	0	2	4	0	4	f	f	f	f	2	0
<b>Byte no</b>	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8

Figure 11 shows a few custom parameters that we have created to display our progress in decoding the tire status.

### Custom Data Parameters

<input checked="" type="checkbox"/>	[ cp1 ]	PGN 65268	Units	✕
<input checked="" type="checkbox"/>	[ cp2 ]	Byte 5	Units	✕
<input checked="" type="checkbox"/>	[ cp3 ]	SPN 1698	Units	✕
<input checked="" type="checkbox"/>	[ cp4 ]	Tire Status	Units	✕

[+ Add Parameter](#)
[\[Help\]](#)
Save Changes

Figure 11 - Custom Parameters Created

First, we need to isolate byte 5 and so we use the *Parse* function with start character 8, length 2, base 16 (hex), and we interpret the data as an unsigned 8 bit char, SQ.U8.

Now we have the byte on its own (0b00000100 = 0x04) and we need to interpret the bits.

bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1
Unused		SPN 1697		SPN 1698		SPN 1699	
0	0	0	0	0	1	0	0

We can either use a bit mask to extract bits 3 and 4 and then compare them with the status options in SPN 1698, or we can use shift operators and a mask to extract the bits as a number. We will use the shift operator and mask.

To get bits 4 and 3 into position 2 and 1, we right shift 2 bits. We then apply a bitwise AND mask to isolate SPN 1698.

	bit 8	bit 7	bit 6	bit 5	bit 4	bit 3	bit 2	bit 1
<b>Byte 5</b>	Unused		SPN 1697		SPN 1698		SPN 1699	
<b>Byte 5</b>	0	0	0	0	0	1	0	0
<b>Byte 5 &gt;&gt; 2</b>	Unused		SPN 1697		SPN 1698		SPN 1699	
<b>Byte 5 &gt;&gt; 2</b>	0	0	0	0	0	0	0	1

**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
15 of 17

<b>Mask (0x03)</b>	0	0	0	0	0	0	1	1
<b>Result</b>	0	0	0	0	0	0	0	1

We have now isolated SPN1698 and can compare it with the status described in the SPN definition.

The script developed is shown below:

```
load('senquip.js');

let PGN65268 = "1032c02404ffff20"; // The PGN of interest - normally this would be
read by the CAN module

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  SQ.dispatch(1, PGN65268);

  let byte5 = SQ.parse(PGN65268, 8, 2, 16, SQ.U8); // extract byte 5
  SQ.dispatch(2, byte5);

  let SPN1698 = (byte5 >> 2) & 0x03; // extract the bits
  SQ.dispatch(3, SPN1698);

  if (SPN1698 === 0) { // compare the SPN with the definition
    SQ.dispatch(4, "Ok");
  }
  else if (SPN1698 === 1) {
    SQ.dispatch(4, "Tire Leak");
  }
  else if (SPN1698 === 3) {
    SQ.dispatch(4, "Error");
  }
  else {
    SQ.dispatch(4, "Unsupported");
  }
}, null);
```

The output from the script is shown in Figure 12 where it can be seen that the value 1 for SPN 1698 has been interpreted as a leak.

<b>Document Number</b> APN0027	<b>Revision</b> 1.1	<b>Prepared By</b> NGB	<b>Approved By</b> NB
<b>Title</b> Bit Manipulation in a Script			<b>Page</b> 16 of 17

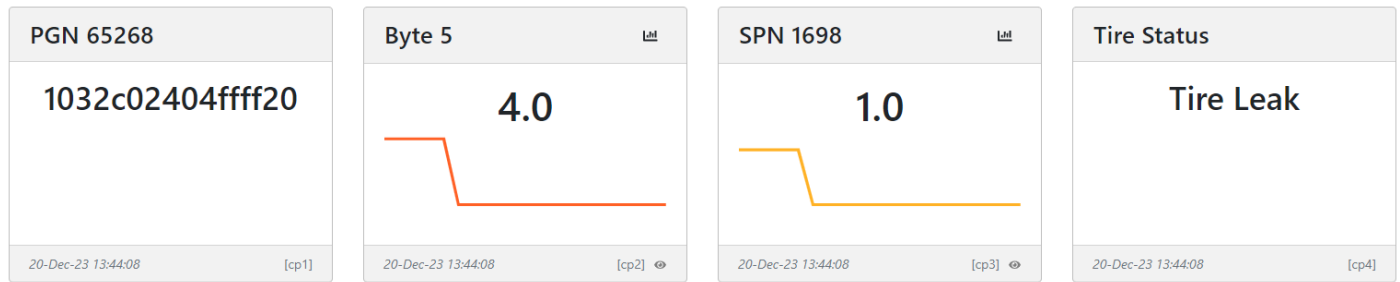


Figure 12 - Output of Script Shown on Senquip Portal

## 11.Extracting PGN from CAN ID

When searching through CAN data is a script, it may be more convenient to search for CAN PGNs rather than identifiers which contain message priority, PGN, and source address. An example of searching through CAN data by looking for identifiers is given in the [Senquip Scripting Guide](#), an excerpt of which is shown below.

```
load('senquip.js');

SQ.set_data_handler(function(data) {
  let obj = JSON.parse(data);

  if (typeof obj.can1 !== "undefined") {
    for (let i = 0; i < obj.can1.length; i++) {

      // Look for the specific PGN:
      if (obj.can1[i].id === 0x18FF1CF2) {
        // Do something with the CAN data
      }
    }
  }
}, null);
```

To extract the PGN from the 29 bit J1939 CAN identifier, we can right shift the identifier by 8 to remove the source address, and then apply a mask to remove the Priority.

29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1
Priority								PGN														Source Address						

The following script show how to implement the bit shift and mask. In the script, we create a function, pgn(), which when given an identifier returns the PGN contained in that identifier. The identifier is passed to the script as a number and so does can be worked with directly without needing conversion.



**Document Number**  
APN0027

**Revision**  
1.1

**Prepared By**  
NGB

**Approved By**  
NB

**Title**  
Bit Manipulation in a Script

**Page**  
17 of 17

```
load('senquip.js');

function pgn(id)
{
    return((id >> 8) & 0x0003FFFF);
}

SQ.set_data_handler(function(data) {
    let obj = JSON.parse(data);

    if (typeof obj.can1 !== "undefined") {
        for (let i = 0; i < obj.can1.length; i++) {

            // Look for the specific PGN:
            if (pgn(obj.can1[i].id) === 0xFEC1) {
                // Do something with the CAN data
            }
        }
    }
}, null);
```

## 12. Conclusions

In the realm of control systems, bitwise operations play a pivotal role in manipulating data at the lowest level of representation. These operations provide a compact and efficient means of handling individual bits within binary data, allowing for precise control over system parameters and status flags. Whether it's encoding and decoding communication protocols, managing hardware registers, or implementing low-level control algorithms, bitwise operations offer a level of granularity that is essential for achieving real-time responsiveness and resource-efficient execution.

The Senquip scripting language provides functions and operators that make testing, setting, clearing, toggling, and isolating bits simple.