UNSW SYDNEY

Introduction to Programmable Logic Controller (PLC) for Assignment 1

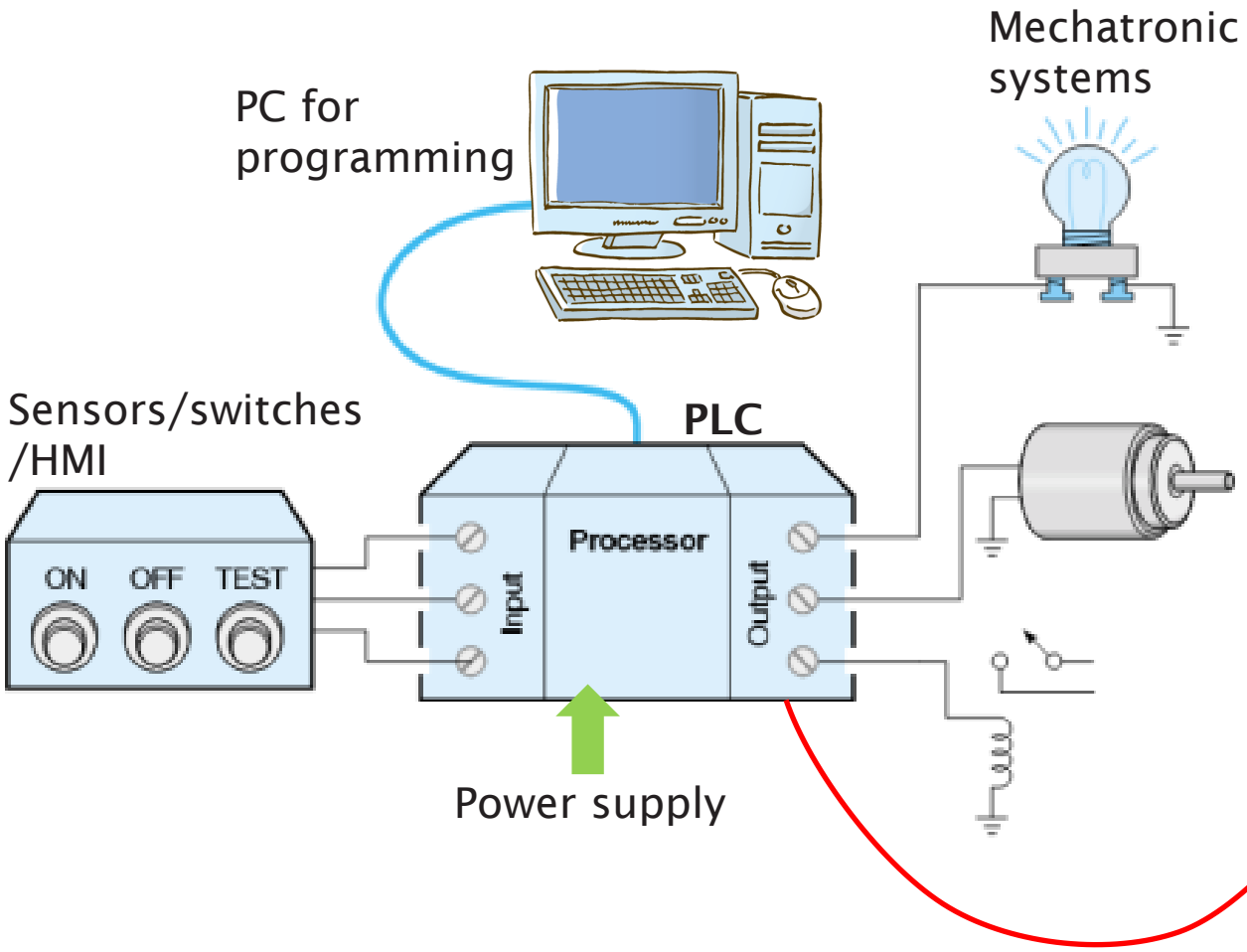
A/Prof Shiyang Tang

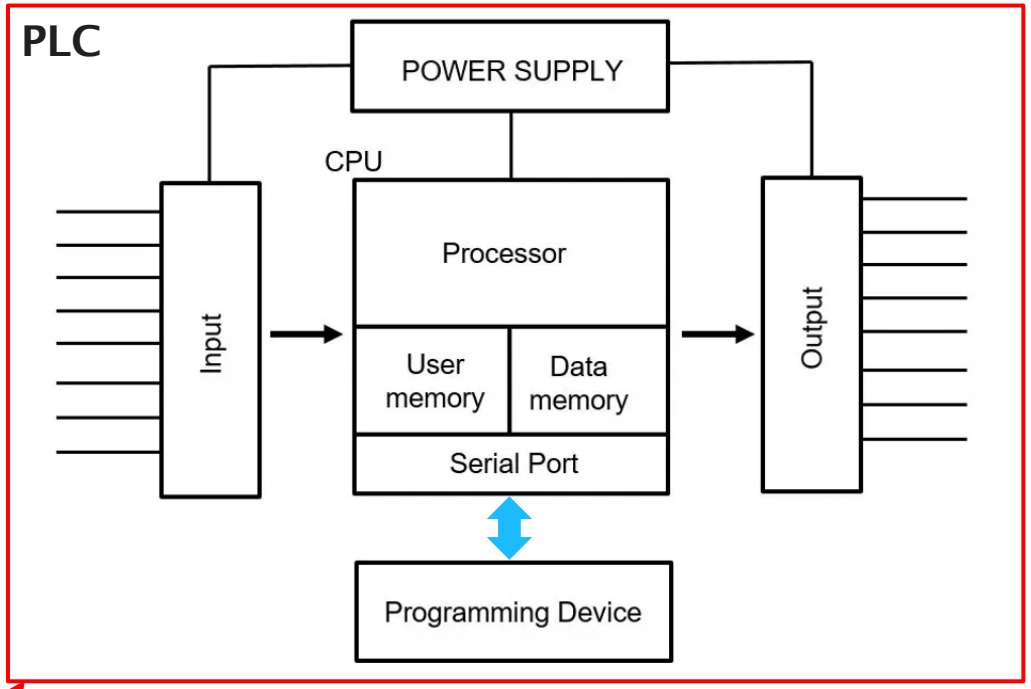
MTRN3500

Computing Applications in Mechatronics Systems

Basic Concepts of PLC

A PLC is an industrial, ruggedised computer designed to monitor inputs, make logic decisions, and drive outputs to control machines and processes in real time.





Why PLC?

PLC vs. microcontrollers (e.g., Arduino)

1. Deterministic control
PLCs: provide predictable scan times and real-time tasking.
Arduino: By writing your own scheduler/interrupt scheme and validating jitter yourself.

2. Rugged, field-ready I/O
PLCs: handle 24 VDC, 0–10 V, 4–20 mA, thermocouples/RTDs, and provide galvanic isolation (i.e. no direct electrical path).
Arduino: every field signal needs a conditioning block (opto-isolators, ADCs/DACs, loop converters, TVS diodes, proper grounding). It's doable, but now you're building a PLC.

3. Safety and compliance
PLCs: safety I/O lets you implement E-Stops, guards, light curtains, STO on drives with diagnostics and certification.
Arduino: solutions rely on external safety relays/controllers.

4. Maintainability & uptime
PLCs: offer forcing, online edits, error codes, and replaceable modules - all standardised.
Arduino: requires your custom tooling and knowledge, increasing MTTR (mean time to repair).

5. Ecosystem integration
PLCs: speak industry's languages out-of-the-box (PROFINET/EtherNet-IP/OPC UA).
Arduino: C/C++ expects custom bridges, protocol glue, and ongoing maintenance.

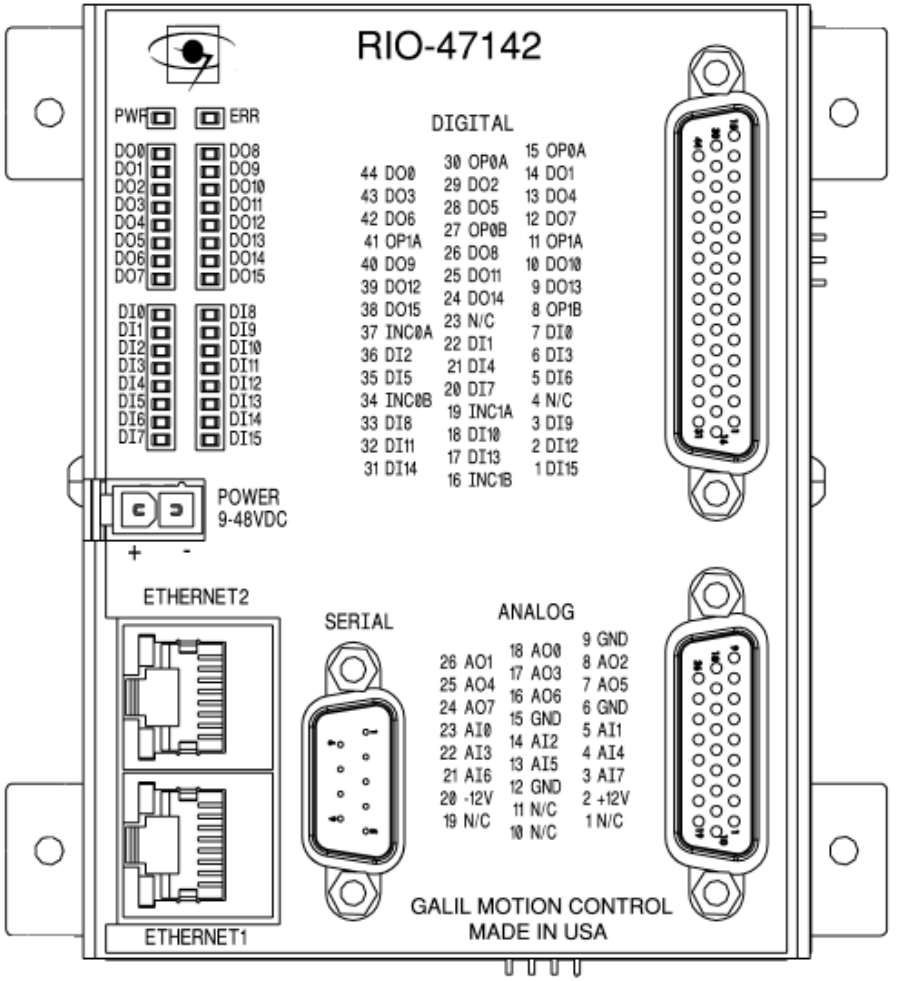
RIO-47142

RIO-47142 is a programmable remote I/O controller built to sit out on the machine. It provides Ethernet-connected digital/analogue I/O with **on-board scripting** (Galil DMC language).

It is not an IEC-61131 PLC, but it plays very well with PLCs over Ethernet.

Key features:

- 32-bit Freescale microcontroller with 32 KB SRAM and 256 KB embedded Flash.
- Dual Ethernet ports with an integrated switch.
- All 16 digital outputs are 500 mA sourcing (high-power).
- No PoE, use an external DC supply.
- Supply range: 9–48 VDC; typical 2.6–4.0 W (no external loads).
- Analogue outputs: ±10 V configurable; 12-bit default, 16-bit optional.
- Analogue inputs: ±10 V configurable; 12-bit default, 16-bit optional.



ADC (Analogue-to-Digital Converter)

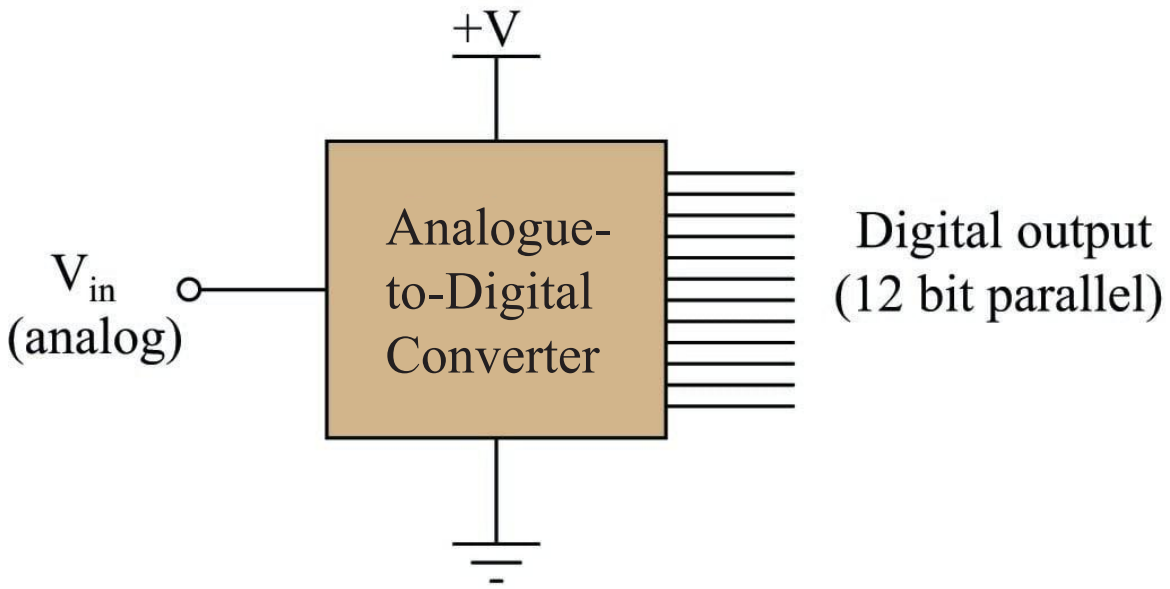
ADC samples a voltage/current and quantises it to a numeric code. ADC on Galil RIO-47142:

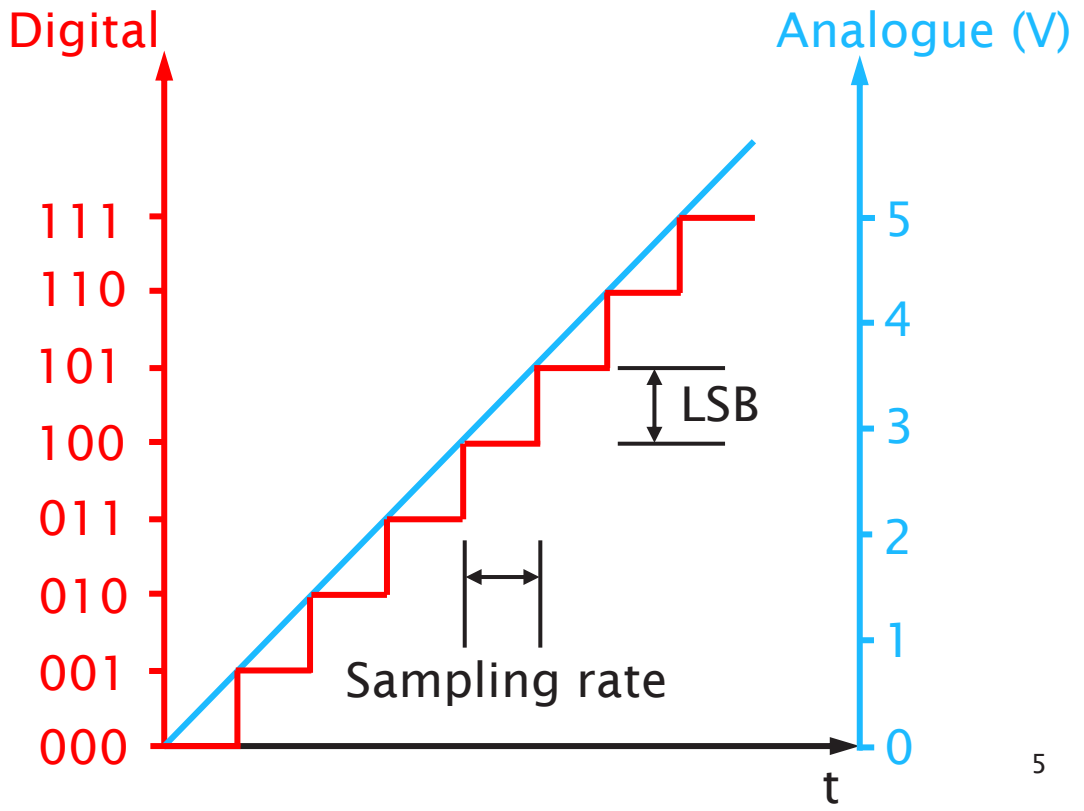
- 0–5 V ADC, 12-bit (≈1.22 mV/LSB); input impedance ~100 kΩ.
- ±10 V configurable ADC (12-bit default, 16-bit optional)

The least significant bit (LSB) in an ADC is the smallest ideal step between two adjacent digital codes. It sets your nominal voltage (or current) resolution. LSB on Galil RIO-47142:

12-bit@0–5 V: LSB = 5 V/2^{12} = 5/4096 ≈ 1.22 mV.

16-bit@±10 V: LSB = 20 V/2^{16} = 20/65536 ≈ 0.305 mV.

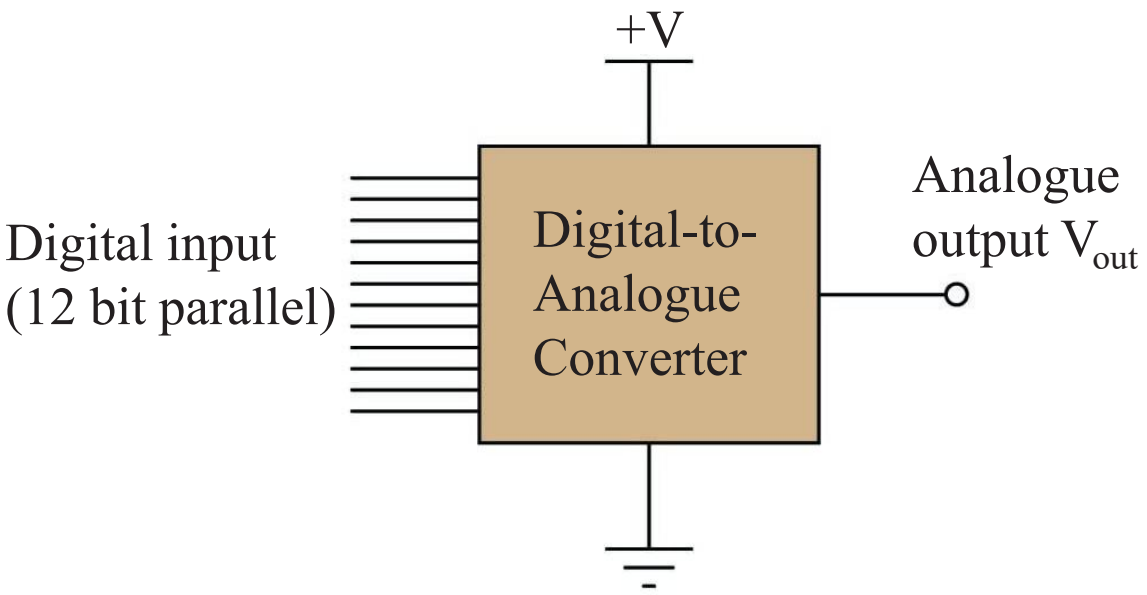


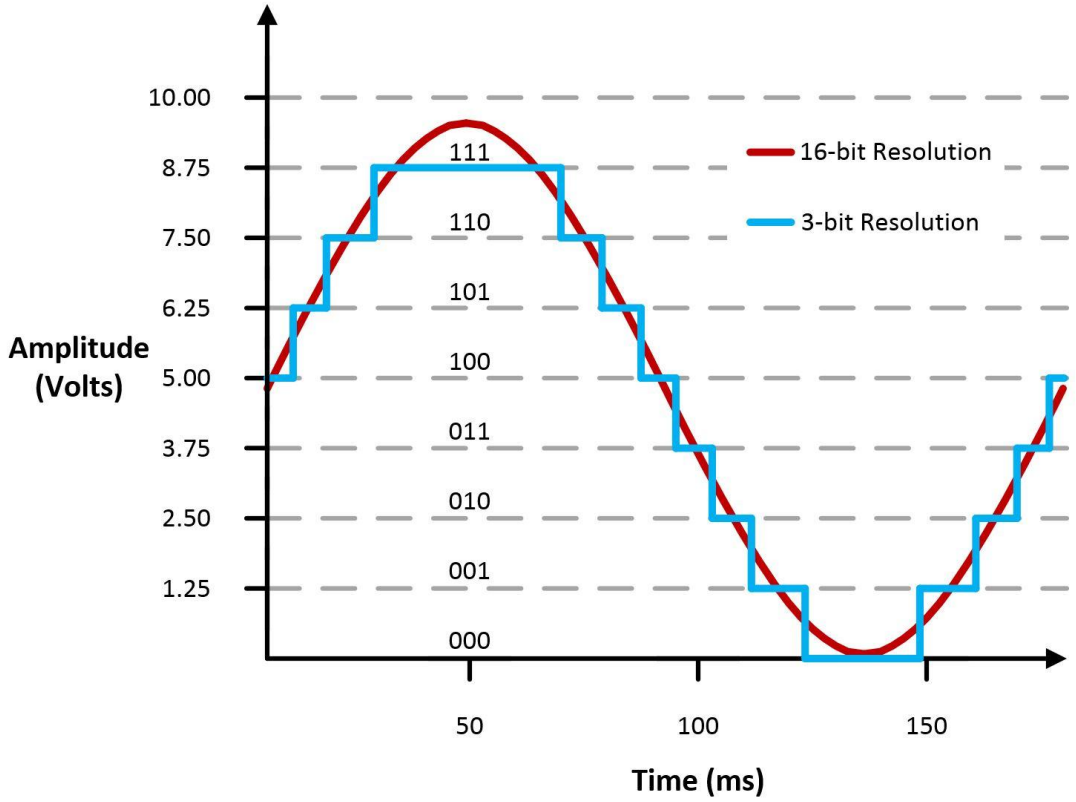


DAC (Digital-to-Analogue Converter)

DAC write a code (or a floating value) and the output drives a voltage. DAC on Galil RIO-47142:

- 0–5 V outputs; 12-bit resolution; max output current 4 mA (source/sink).
- 0–10 V, ±5 V, ±10 V; 12-bit default or 16-bit optional; ±10 V max/min, max output current 4 mA .



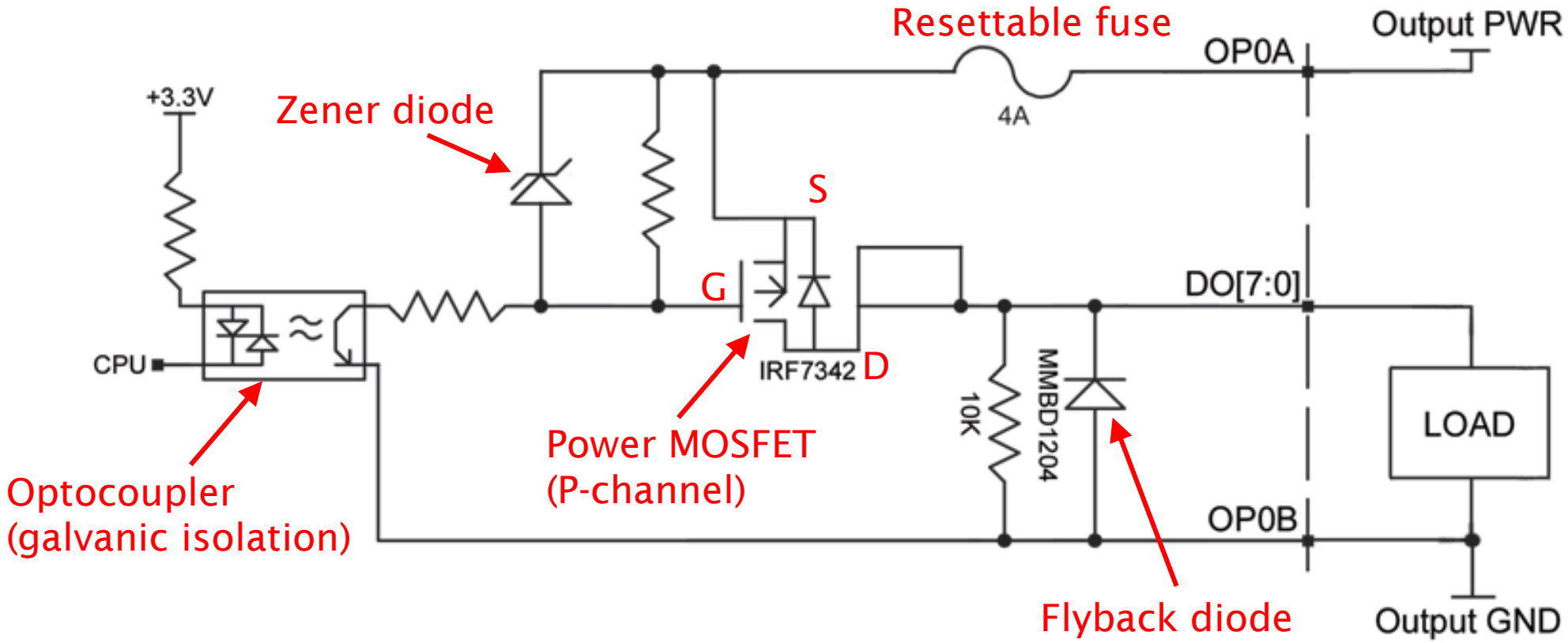


https://hackprojectsindia.blogspot.com/2017/10/dacdigital-to-analog-conversion-using.html

Digital Outputs

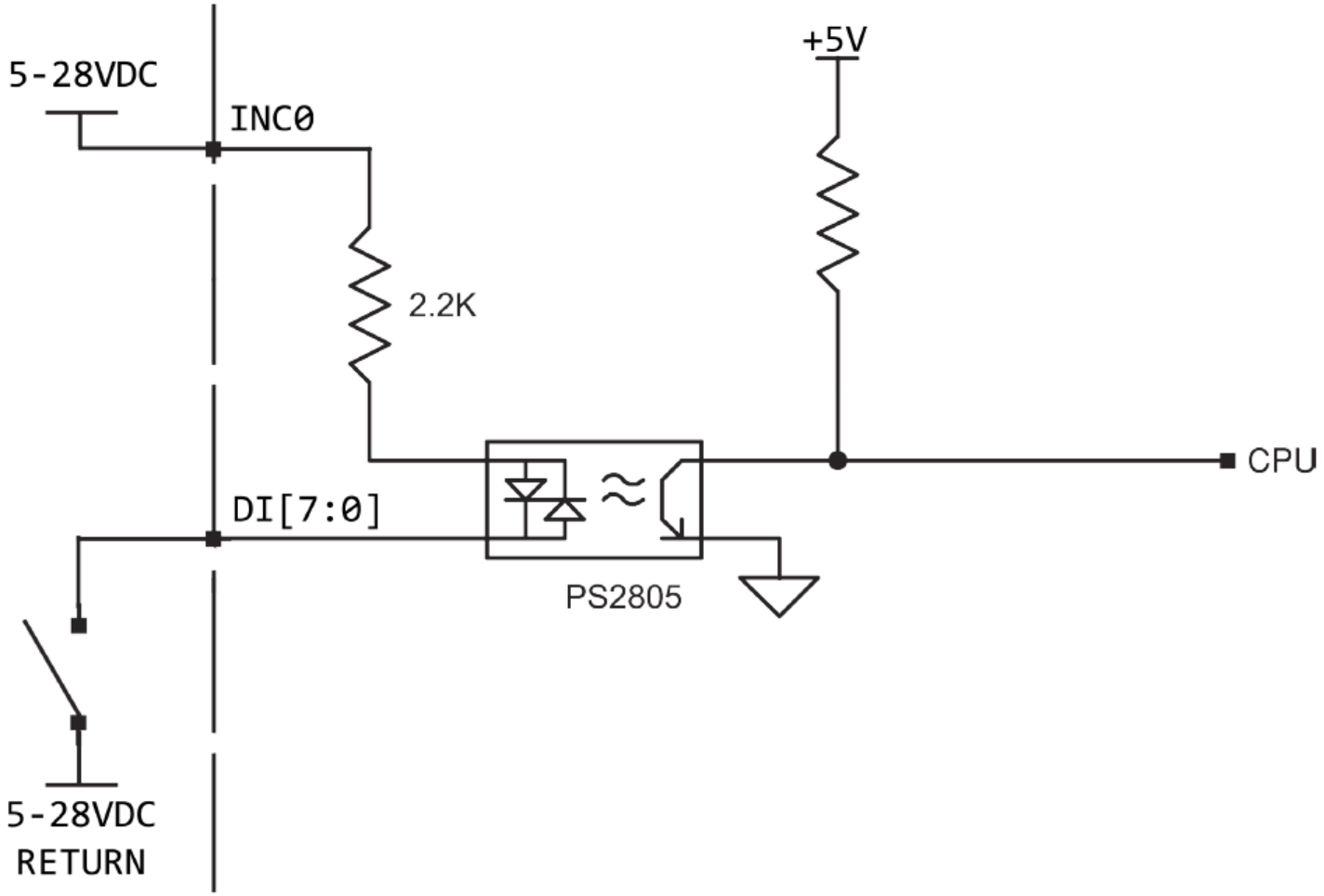
High power sourcing outputs (HSRC): All 16 DOs are HSRC by hardware. They are capable of sourcing up to 500 mA per output and up to 3 A per bank.

The voltage range for the outputs is 12-24 VDC. These outputs are capable of driving inductive loads such as solenoids or relays.



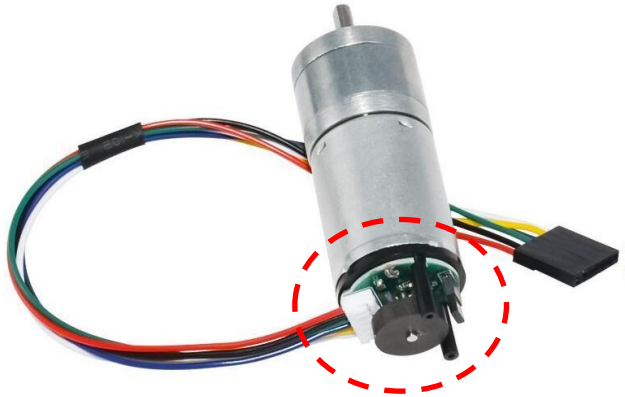
Digital Inputs

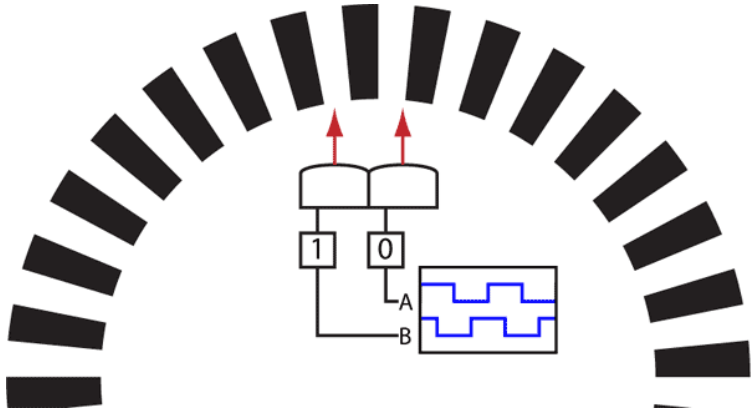
All the digital inputs (labelled DI) are optoisolated and must be powered with a voltage ranging between 5-24 VDC (minimum current to turn on inputs is 1.2 mA).

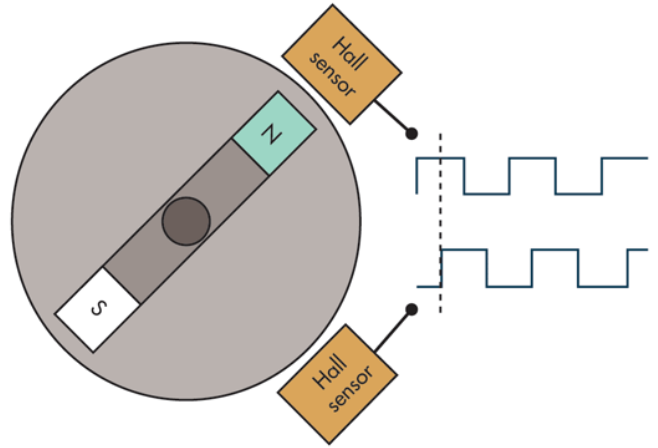


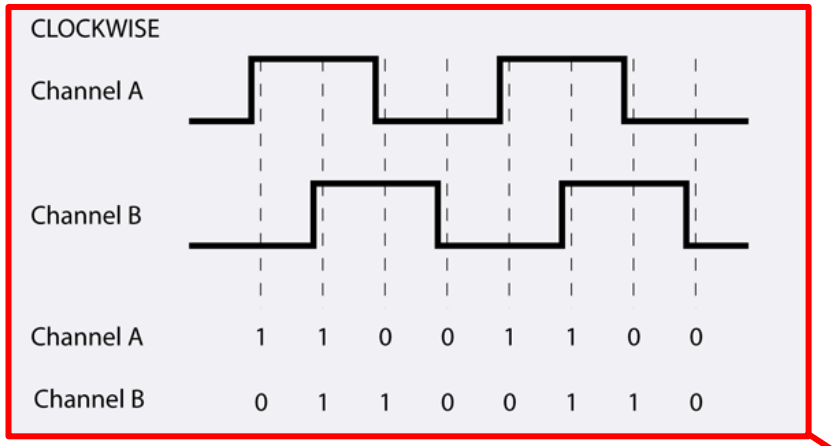
Quadrature Encoder

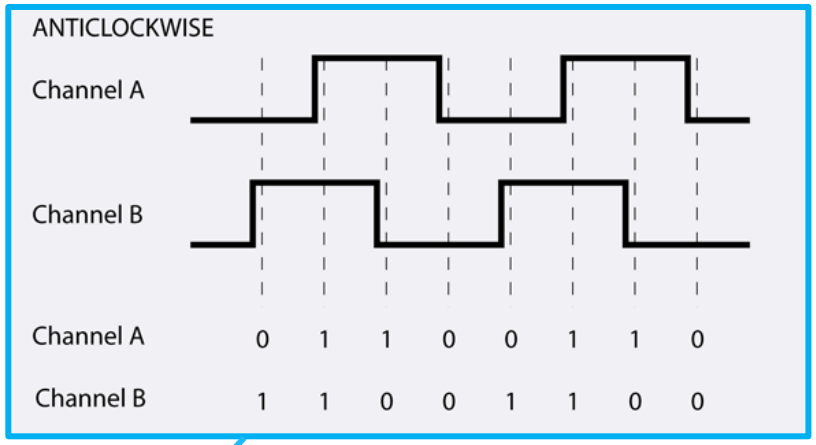
Two square waves A and B, **90**˚ out of phase. The transition order tells direction; each transition is a count. Quadrature has four valid logic states: 00→01→11→10→00 (forward) or the reverse (backward).

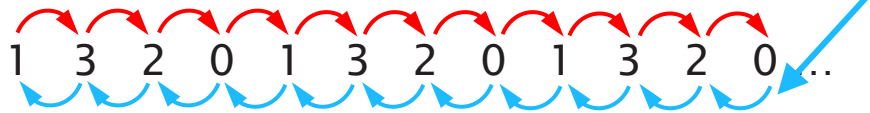








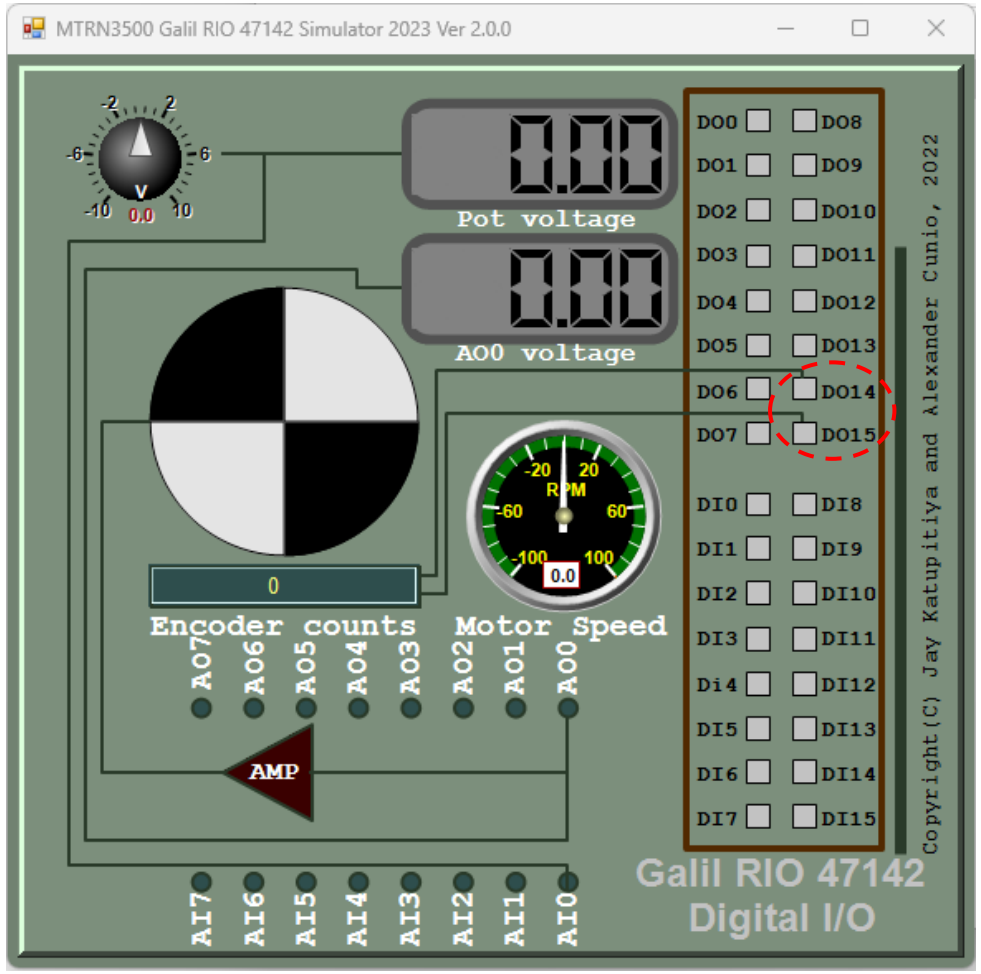




Quadrature Encoder in Galil

All Encoder Options will utilise Digital Inputs 12,13,14, and 15; and Digital Outputs 12,13,14 and 15.
Single-ended encoders: connect to A+ / B+ and leave A− / B− floating.

Encoder Signal	Label (Connector.Pin)
Channel 0 A+	DO14 (J4.24)
Channel 0 A-	DO12 (J4.39)
Channel 0 B+	DO15 (J4.38)
Channel 0 B-	DO13 (J4.9)
Channel 1 A+	DI14 (J4.31)
Channel 1 A-	DI12 (J4.2)
Channel 1 B+	DI15 (J4.1)
Channel 1 B-	DI13(J4.17)
Ground	N/C (J4.41)



Bit Operations

Decimal (base-10)

Digits: 0 1 2 3 4 5 6 7 8 9.

Example: $173 = 1 \times 10^2 + 7 \times 10^1 + 3 \times 10^0$

C/C++ literal prefixes: n/a

Binary (base-2)

Digits: 0 1; each place is a **bit**.

Example: $1010_2 = 1 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 10_{10}$

C/C++ literal prefixes: 0b (e.g., 0b1010)

Octal (base-8)

Digits: 0 1 2 3 4 5 6 7; each octal digit ≡ 3 bits.

Example: $0755_8 = 7 \times 8^2 + 5 \times 8^1 + 5 \times 8^0$

C/C++ literal prefixes: 0 (e.g., **0**755, leading zero means octal)

Hexadecimal (base-16)

Digits: 0 1 2 3 4 5 6 7 8 9 A B C D E F; each hex digit ≡ **4 bits (a nibble)**.

Example: $0xAB2F = 10 \times 16^3 + 11 \times 16^2 + 2 \times 16^1 + 15 \times 16^0$

C/C++ literal prefixes: 0x (e.g., 0xFF)

Bit Operations

Converting binary ↔ hex

Principle: 1 hex digit ≡ 4 binary bits. Group binary in 4s from the right; map each group to 0–F.

Bin	Hex	Bin	Hex
0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Binary → Hex:

$1111\ 1110_2$ → 0xFE (because 1111 = F and 1110 = E)

$1101\ 0110_2$ → 0xD6

Hex → Binary:

0x3A7F → 0011 1010 0111 1111

Octal relation: group bits in 3s (since $2^3=8$)

$101\ 101\ 110_2$ → $5\ 5\ 6_8$ → octal 0556

Bit Operations

Two-input logic: AND, OR, and XOR

A	B	A AND B	A OR B	A XOR B
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Single-input logic: NOT

X	NOT X
0	1
1	0

Bit Operations

Bit shifts

C/C++: “<<“ left shift; “>>” right shift

e.g., left shift by 2 (move all bits two places left; zeros fill on the right):

```
        0001 0110   (0x16 = 22)
<< 2
=       0101 1000   (0x58 = 88)
```

e.g., right shift by 3 (move three places right; zeros fill on the left for unsigned):

```
        0001 0110   (0x16 = 22)
>> 3
=       0000 0010   (0x02 = 2)
```

Bit Operations

Set a bit high (OR with a mask)

C/C++: "|" OR operator

e.g., set bit 5 to 1 (note that bit positions are zero-indexed from the right):

```
        0101 0010
|       0010 0000   (mask)
=       0111 0010
```

Clear a bit low (AND with the inverted mask)

C/C++: "&" AND operator

e.g., clear bit 2 to 0:

```
        0111 0100
&       1111 1011   (inverted 0000 0100 mask)
=       0111 0000
```

Bit Operations

Toggle a bit (XOR with a mask)

C/C++: "^" XOR operator

e.g., flip bit 0:

```
        0111 0000
^       0000 0001   (mask)
=       0111 0001
```

e.g., flip bit 4:

```
        0111 0000
^       0001 0000   (mask)
=       0110 0000
```

Test a bit (AND with a mask, check the result)

e.g., is bit 7 set in 0x71?

```
        0111 0001   (0x71)
&       1000 0000   (mask)
=       0000 0000   (bit 7 is not set)
```

e.g., is bit 4 set?

```
        0111 0001   (0x71)
&       0001 0000   (mask)
=       0001 0000   (bit 4 is set)
```

Galil Programming Guide

RIO-47142 understands **short text commands** you send over Ethernet (TCP). Each command ends with a semicolon ";". You can:

- **Act** (change outputs, set voltages, zero an encoder)
- **Ask** (read inputs, voltages, positions)

OP ***Output Port***

OP $n_0, n_1, n_2, n_3, n_4, n_5$

Usage	OP n ...	Arguments specified with an implicit, comma-separated order
Operands	_OP0 _OP1 _OP2 _OP3 _OP4 _OP5	Operand holds the value last set by the command

Description

The OP command sets the output ports of the controller in a bank using bitmasks. Arguments to the OP command are bit patterns (decimal or hex) to set entire banks (bytes) of digital outputs. Use SB, CB or OB to set bits individually.

Arguments

Argument	Min	Max	Default	Resolution	Description	Notes
n_0	0	255	0	1	Decimal representation: Outputs 0-7	
n_1	0	255	0	1	Decimal representation: Outputs 8-15	
n_2	0	255	0	1	Decimal representation: Outputs 16-23	RIO-473xx
n_3	0	255	0	1	Decimal representation: Outputs 24-31	RIO-473xx with -24ExOut option
n_4	0	255	0	1	Decimal representation: Outputs 32-39	RIO-473xx with -24ExOut option
n_5	0	255	0	1	Decimal representation: Outputs 40-47	RIO-473xx with -24ExOut option

"**OP**" sets the digital outputs by giving two numbers (note that RIO-47142 only has 2 output banks, i.e. 16 outputs):

- The first number is the low bank (outputs 0–7).
- The second number is the high bank (outputs 8–15).

e.g. turn ON outputs 0–7 and 12-15, and the rest are OFF

Command you send:

"**OP 255, 240;**"

or

"**OP $FF, $F0;**"

Why: 255 = 1111 1111= 0xFF, 240 = 1111 0000 = 0xF0 ($ represents hex in Galil commands).

@AN *Analog Input Query*

@AN[n]

Usage	variable = @AN[value]	Performs a function on a value or evaluated statement and returns a value

Description

The @AN[] operator returns the value of the given analog input in volts.

Arguments

Argument	Min	Max	Default	Resolution	Description	Notes
n	0	7	N/A	1	Analog input to query	

Remarks

- @AN[] is an operand, not a command. It can only be used as an argument to other commands and operators

e.g. read an analogue input from Analogue Input 2

Command you send:

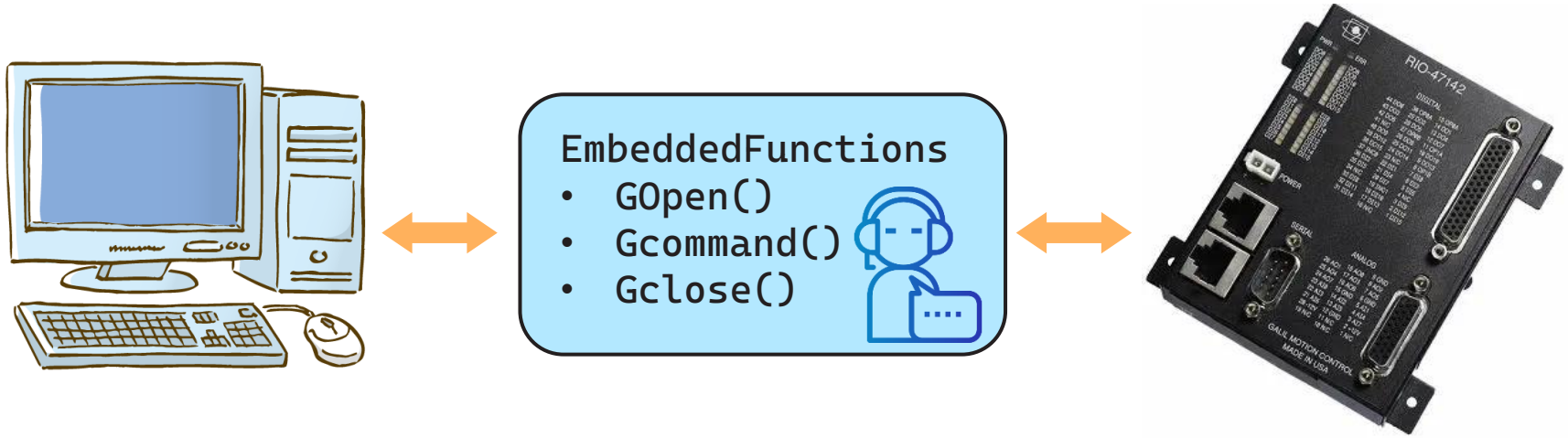
"**MG @AN[2] ;**" (print analogue input 2)

"**x = @AN[2] ;**" (assign analogue input 2 to a variable x)

The "MG" command is used to send strings, operands, variables, and array values to a specified destination.

Galil Programming Guide

How do you establish the connection with your Galil RIO-47142 and send commands (or receive messages) using your PC?



Use `EmbeddedFunctions`, which can be regarded as the tiny "phone operator" between your PC app and the Galil.

It opens the line, sends a command text, waits for the reply text, and hangs up when you are done. This is achieved through 3 functions:

- `GOpen(IP)`: "Connect to the Galil at this IP address."
- `GCommand(…)`: "Send command string; give me the reply (if any)."
- `GClose()`: "Disconnect cleanly."

Galil Programming Guide

```
GReturn GOpen(GCStringIn address, GCon* g);
```

"Connect to the controller." `gclib` parses the IP address (**192.168.0.120**), opens the underlying transport (TCP/serial/etc.), performs a short handshake, and hands you back an opaque connection handle (`GCon`) in `*g` that you'll use for all further calls.

`Greturn:` Status/return type used by Galil's `gclib` C API; it is an integer that tells you whether a library call succeeded or failed.

`Gcon:` A *connection handle* used by Galil's official `gclib` API.

It is a small, library-assigned identifier for a live connection. It points to all the state the library keeps internally (socket, buffers, timeouts, etc.). You use it so the library can route your command to the right device without you managing sockets yourself.

- **Create**
  `GOpen(address, &g)` → on success, `g` is a valid handle.
- **Use**
  `GCommand(g, "…", buf, len, &n)`: send/receive as much as you like
  `GInfo(g, info, info_len)`
- **Destroy**
  `GClose(g)` → releases resources; do not use `g` after this.

```
GReturn GCommand(GCon g, GCStringIn command, GBufOut buffer, GSize buffer_len, GSize* bytes_returned);
```

“Send one command, get one reply.” Synchronously writes an ASCII DMC command to the controller and fills your buffer with the reply text (if any).

- `g` — open connection handle from `GOpen`.
- `command` — null-terminated ASCII, e.g. "OP 255, 240" or "MG @AN[2]".
- `buffer` — caller-supplied memory where the **reply text** will be written.
- `buffer_len` — size of buffer in bytes.
- `bytes_returned` — out-parameter for the number of meaningful bytes written to the buffer.

Action commands (e.g., OP, SB, CB, AO, WE) usually return no numeric text; you still receive a short protocol termination that the library consumes.

Interrogation commands (e.g., MG ..., QE, @AN[...]) return a line of text such as "2.5034".

```
GReturn GInfo(GCon g, GCStringOut info, GSize info_len);
```

Fills info with a short, human-readable string describing the connection and device/library.

```
GReturn GClose(GCon g);
```

Cleanly closes the session and releases resources. After this, g is no longer valid.

Galil Programming Guide

UNSW SYDNEY

Ethernet communication example

```
#using <System.dll>
#include <cstdint>  //Brings fixed-width integer types (e.g., uint8_t).

using namespace System;
using namespace System::Net::Sockets;
using namespace System::Text;

int main()
{
    TcpClient^ GalilMngHndl;                //Declares a managed handle to a .NET TCP client.
    String^ IPAddress = "192.168.0.120";    //IP address for Galil RIO-47142.
    int Port = 23;                          //TCP port 23 (Telnet), a standard port for Galil ASCII command/response.
    array<uint8_t>^ SendData;               //Managed byte arrays for outgoing data.
    array<uint8_t>^ RecvData;               //Managed byte arrays for incoming data.
    String^ Command;                        //Declare a string handle for Command.
    String^ Response;                       //Declare a string handle for Response.

    GalilMngHndl = gcnew TcpClient(IPAddress, Port);        //Creates and connects a TCP socket to 192.168.0.120:23.
    GalilMngHndl->SendBufferSize = 64;                      //Hints to the underlying socket buffer sizes.
    GalilMngHndl->ReceiveBufferSize = 2048;
    GalilMngHndl->SendTimeout = 500;                        //I/O timeouts in milliseconds.
    GalilMngHndl->ReceiveTimeout = 500;
    GalilMngHndl->NoDelay = true;                           //Disables Nagle's algorithm so short commands go out immediately.
    NetworkStream^ GalilStream = GalilMngHndl->GetStream(); //Gets a stream wrapper over the socket.
    SendData = gcnew array<uint8_t>(64);                    //Allocates the actual buffer for the data to be sent.
    RecvData = gcnew array<uint8_t>(2048);                  //Allocates the actual buffer for the data to be received.

    Command = "MG @AN[2]\r";                        //Galil command: print the value of analog input 2; ends with \r so the controller executes the line.
    SendData = Encoding::ASCII->GetBytes(Command);  //Encodes the command string to raw ASCII bytes to send over TCP.
    GalilStream->Write(SendData, 0, SendData->Length);  //Sends the command bytes.
    Threading::Thread::Sleep(10);

    GalilStream->Read(RecvData, 0, RecvData->Length);   //Reads up to 2048 bytes into RecvData.
    Threading::Thread::Sleep(25);
    Response = Encoding::ASCII->GetString(RecvData);    //Decodes the RecvData array to text. Need to decode the whole 2048 every time, not efficient.
    Console::WriteLine(Response);                       //Prints decoded message.
    return 0;
}
```

Appendix 1: NetworkStream::Write

```
SendData = Encoding::ASCII->GetBytes(Command);
GalilStream->Write(SendData, 0, SendData->Length);
```

```
Write(array<Byte>^ buffer, int offset, int count)
```

- **SendData (buffer)**
  The byte array to send. It contains the ASCII bytes for the command (e.g., M G @ A N [2] \r).
- **0 (offset)**
  The starting index in SendData to begin sending from. 0 means “start at the first byte of the array.” If non-zero, it skips that many leading bytes and starts in the middle of the array.
- **SendData->Length (count)**
  The number of bytes to send, beginning at offset. Using SendData->Length with offset 0 means “send the entire array.”