Application Note

2500 Series®

Programmable Automation Control System

Using a CTI 2500P-ACP1 Application Co-Processor module for RS232 serial data acquisition from an Ohaus IP Series High-Capacity Precision Top-loader bench scale

Scope:

This application is an update to a previous Tech Note on using the CTI 2572-TCM2 Serial Interface Adapter for this application. The 2500P-ACP1 is a more powerful and flexible platform which makes this application simpler to develop, test and deploy. NOTE: This application can be run on a 2500P-JACP Application Co-Processor module as well, with a change in the configuration of the PLC interface.

2500P-ACP1 Description:

The 2500P-ACP1 module is a general-purpose auxiliary controller that enhances the capabilities of all CTI 2500 Series® and SIMATIC® 505 PLC systems. This Advanced Function Module includes high-speed processing and multi-protocol communications support to provide existing systems with a significant increase in performance, features, and functionality. The 2500P-ACP1 runs as a PLC coprocessor performing complex logic/math functions, data logging, and communications with external devices. Although the 2500P-ACP1 can operate as a standalone controller, the application generally requires data transfer between a host PLC and the module.

Two different data transfer options are provided:

PLC I/O (2500P-ACP1 only):

The 2500P-ACP1 emulates a standard I/O module configured as 32WX / 32WY and/or 32X / 32Y image register data points. This allows the module to work with SIMATIC® 545/555 CPUs in limited applications where a maximum of 32 words of data is transferred to/from the CPU each PLC scan.
****(This is the transfer mode chosen for this application)*

Data Cache (2500P-ACP1 and 2500P-JACP):

Proprietary link offering enhanced data throughput to CTI 2500 Series® controllers via a dedicated Ethernet connection. Supports up to 4096 variables mapped to any PLC memory type (including Loop/Alarm variables). The 2500P-ACP1 includes two external 10/100Mb Ethernet ports with automatic detection of network speed, duplex mode, and cable wiring.

Block Transfer (2500P-JACP only):

The Block Transfer driver provides a method to transfer large blocks of data between a Janus Application Coprocessor (JACP) module and a SIMATIC® 545/555 PLC or CTI 2500 Series PLC via the I/O backplane. This driver provides significantly greater communications capabilities compared to "PLC I/O" above. This driver supports up to 4096 variables mapped to any PLC memory type (including Loop/Alarm variables).

Serial Port:

A serial port (male DB-9) provides an electrical interface for RS-232-C (subset) and RS-422-A connections. All port parameters are set by software configuration. Sending and receiving of messages is controlled by program logic.

NOTE:

In the Appendix at the end of this Tech Note you will find additional details on the ACP1, as well as descriptions of all of the software instructions used and other pertinent information.

ACP1 solution scope:

The following describes the means by which the 2500P-ACP1 will read the weight value from the scale.

SEND data by simulating the scale output:

This application uses an ACP1 to simulate the known ASCII data stream from the Ohaus scale.
This ACP1 module performing the SEND operation will be referred to as **ACP1**
This WorkBench application is ‘ACP1_Serial_2’.
The **ACP1** module is installed in a base with a CTI 2500-C400 PLC

RECEIVE data:

A 2nd ACP1 module receives and decodes the ASCII data stream to read the weight value from the scale and write it to a PLC memory location.

This ACP1 performing the RECEIVE operation will be referred to as **ACP2**
This WorkBench application is ‘ACP2_Serial’

The **ACP2** module is installed in a base with a Simatic 555-1106 PLC

This program uses ‘ACP1 I/O Interface’ to send the following to the PLC:

Value of ‘Decode_done’ bit sent to PLC X2001; this bit is an error indication.
In the PLC RLL this input is used to stroke a counter to track decoding errors.
This bit could also be used to signal an HMI to a communications problem.

Value of ‘Scale_wt_intX10’ word sent to PLC WX2065.
In the PLC SFPGM1 the value in WX2065 is multiplied times 10 and put into V1000. (Real).
This is the actual scale weight as a floating point number.

Communications wiring:

The field wiring part of this ACP1 module communication is achieved by plugging a standard Null Modem cable into the DB9 serial port on each module.

Ohaus scale configuration:

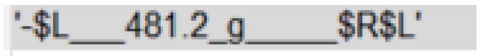
- Continuous print mode
- 9600 baud
- No parity.

Sending / Encoding and Receiving / Decoding the ASCII data stream

Using an ACP1 to simulate the serial data from the bench scale:

Number value to be transmitted = -481.2

Sent ASCII string from ACP1:



'-$L___481.2_g_____$R$L'

'-$L___481.2_g_____$R$L'

Where:

1st character = this field represents the polarity of the number value transmitted.

This field is a Minus Sign '-' indicating a negative number.

If this field were a positive number this 1st character would have been an Underscore '_',

that ASCII string would then be '_$L___481.2_g_____$R$L'

2nd character = '$L' this is a 'Line Feed'.

3rd, 4th, and 5th characters = '_' which are Underscores.

6th, 7th, 8th, 9th and 10th characters = '481.2' is the actual number value transmitted in the format 'xxx.x'.

11th and 12th characters = '_g' which are an Underscore and 'g' (for grams).

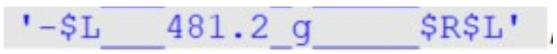
13th – 17th characters = '_' are Underscores.

18th character = '$R' = this is a Carriage Return.

19th character = '$L' = this is a Line Feed.

Using an ACP1 to receive and decode the serial data from the bench scale:

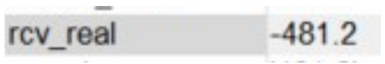
Received ASCII string at ACP2:



```
'-$L___481.2_g_____$R$L'
```

'-$L___481.2_g_____$R$L'

Decoded number value:



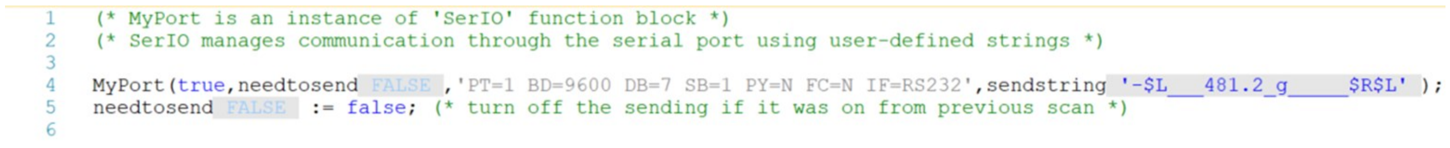
rcv_real	-481.2

ACP1 = SEND

This 1st ACP is being used to send data out the serial port to be received by the 2nd ACP. The screen captures below show the programming requirements to configure the serial port and to send the data out the serial port. The data sent matches the format explained on Page 4. For testing, the only part that you should modify is the actual scale weight embedded in this message.

Configure Port:

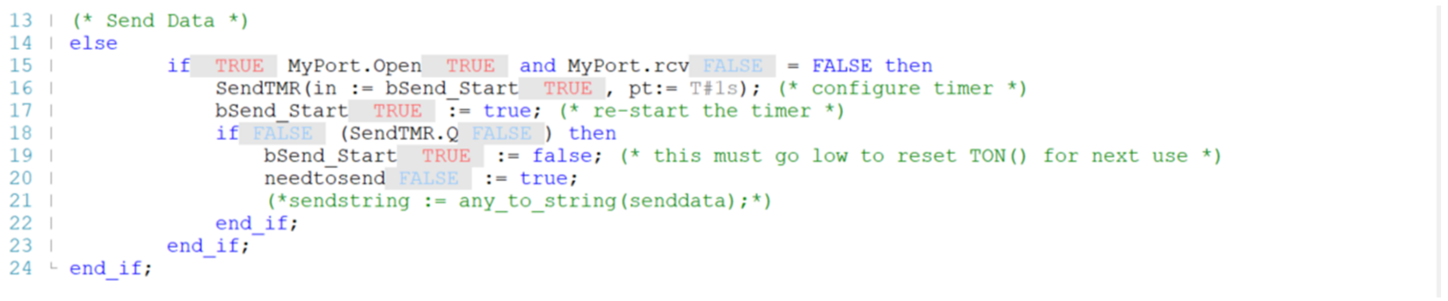
Note: these communications parameters match the specifications for the Ohaus bench scale.



```
(* MyPort is an instance of 'SerIO' function block *)
(* SerIO manages communication through the serial port using user-defined strings *)

MyPort(true,needtosend FALSE ,'PT=1 BD=9600 DB=7 SB=1 PY=N FC=N IF=RS232',sendstring '-$L___481.2_g_____$R$L' );
needtosend FALSE  := false; (* turn off the sending if it was on from previous scan *)

```

Send Data:

Note: this data send operation is unsolicited and is cyclic based on the timer function.



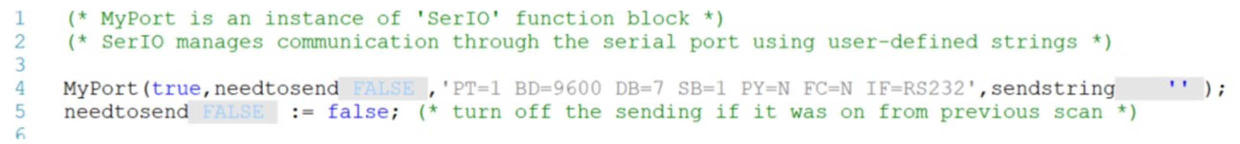
```
(* Send Data *)
else
        if  TRUE  MyPort.Open  TRUE  and MyPort.rcv FALSE  = FALSE then
            SendTMR(in := bSend_Start  TRUE , pt:= T#1s); (* configure timer *)
            bSend_Start  TRUE  := true; (* re-start the timer *)
            if FALSE  (SendTMR.Q FALSE ) then
                bSend_Start  TRUE  := false; (* this must go low to reset TON() for next use *)
                needtosend FALSE  := true;
                (*sendstring := any_to_string(senddata);*)
            end_if;
        end_if;
end_if;
```

ACP2 = RECEIVE

This 2nd ACP is being used to receive data coming into the serial port from the 1st ACP. The screen captures below show the programming requirements to configure the serial port and to receive the data from the serial port. The data received matches the format explained on Page 4. In the following pages this data will be decoded to extract the scale weight itself.

Configure Port:

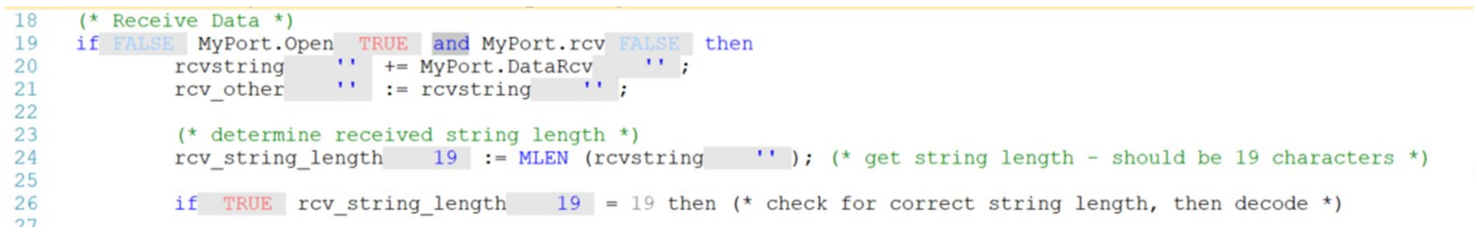
Note: these communications parameters match the specifications for the Ohaus bench scale.



```
 (* MyPort is an instance of 'SerIO' function block *)
 (* SerIO manages communication through the serial port using user-defined strings *)

MyPort(true,needtosend FALSE ,'PT=1 BD=9600 DB=7 SB=1 PY=N FC=N IF=RS232',sendstring    '' );
needtosend FALSE  := false; (* turn off the sending if it was on from previous scan *)
```

Receive Data:

Note: this ASCII data is tested to determine if the string length is correct.



```
 (* Receive Data *)
if FALSE  MyPort.Open  TRUE  and MyPort.rcv FALSE  then
        rcvstring    ''  += MyPort.DataRcv    '' ;
        rcv_other    ''  := rcvstring    '' ;

        (* determine received string length *)
        rcv_string_length    19  := MLEN (rcvstring    '' ); (* get string length - should be 19 characters *)

        if  TRUE  rcv_string_length    19  = 19 then (* check for correct string length, then decode *)
```

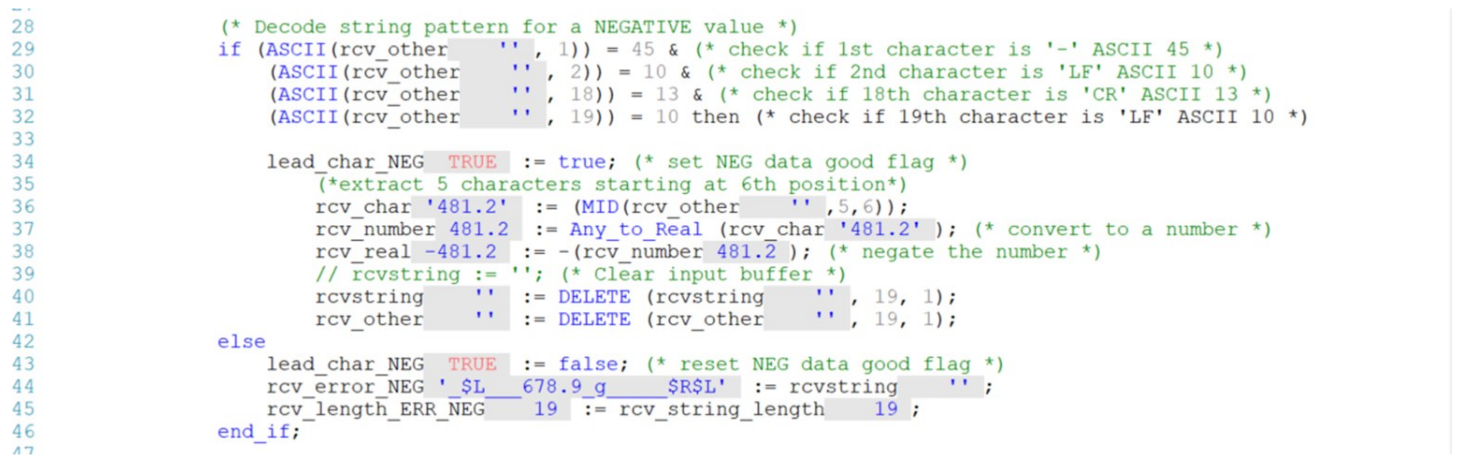
Number value received = *-481.2*

ACP2 = RECEIVE

This 2nd ACP is being used to receive data coming into the serial port from the 1st ACP. This screen captures below show the programming requirements to configure the serial port and to received the data from the serial port. The data received matches the format explained on Page 4. In the following pages this data will be decoded to extract the scale weight itself.

Analyze and decode the ASCII string if it is a negative number

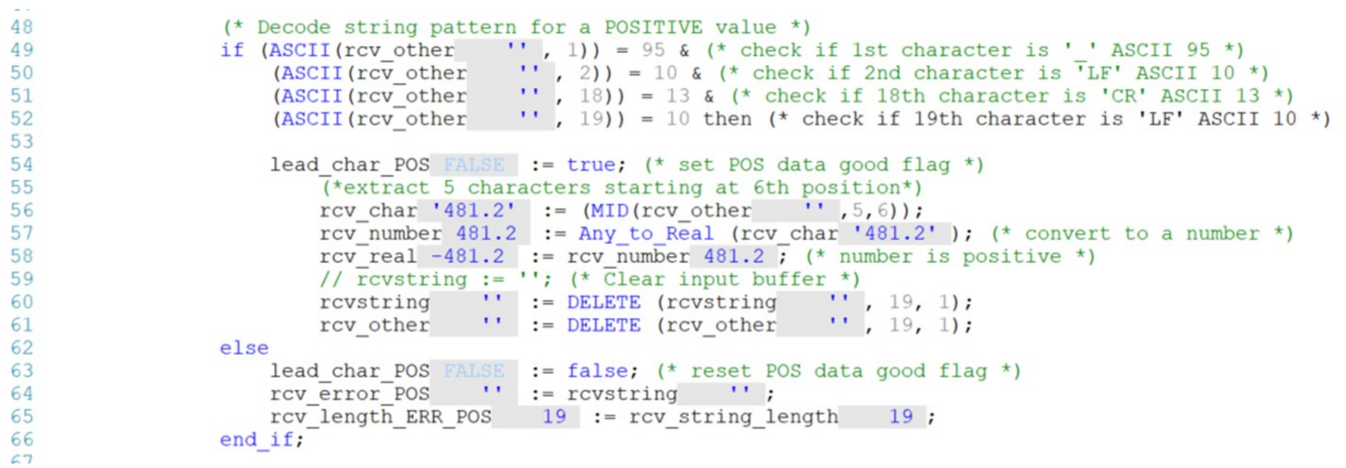
Note: this data packet has two leading characters and two trailing characters all of which aid in determining that this packet is complete and also the polarity of the number value.



```
(* Decode string pattern for a NEGATIVE value *)
if (ASCII(rcv_other    '' , 1)) = 45 & (* check if 1st character is '-' ASCII 45 *)
    (ASCII(rcv_other    '' , 2)) = 10 & (* check if 2nd character is 'LF' ASCII 10 *)
    (ASCII(rcv_other    '' , 18)) = 13 & (* check if 18th character is 'CR' ASCII 13 *)
    (ASCII(rcv_other    '' , 19)) = 10 then (* check if 19th character is 'LF' ASCII 10 *)

    lead_char_NEG  TRUE  := true; (* set NEG data good flag *)
        (*extract 5 characters starting at 6th position*)
        rcv_char '481.2'  := (MID(rcv_other    '' ,5,6));
        rcv_number 481.2  := Any_to_Real (rcv_char '481.2' ); (* convert to a number *)
        rcv_real -481.2  := -(rcv_number 481.2 ); (* negate the number *)
        // rcvstring := ''; (* Clear input buffer *)
        rcvstring     ''  := DELETE (rcvstring     '' , 19, 1);
        rcv_other     ''  := DELETE (rcv_other     '' , 19, 1);
else
    lead_char_NEG  TRUE  := false; (* reset NEG data good flag *)
    rcv_error_NEG '_$L___678.9_g_____$R$L'  := rcvstring    '' ;
    rcv_length_ERR_NEG    19  := rcv_string_length    19 ;
end_if;
```

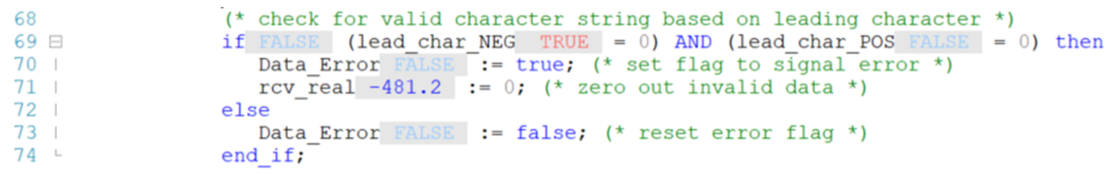
Analyze and decode the ASCII string if it is a positive number

Note: this data packet has two leading characters and two trailing characters all of which aid in determining that this packet is complete and also the polarity of the number value.



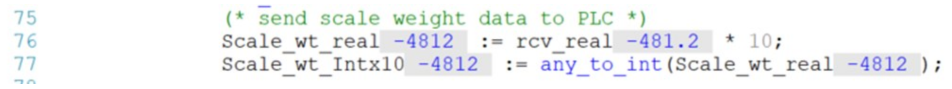
```
(* Decode string pattern for a POSITIVE value *)
if (ASCII(rcv_other    '' , 1)) = 95 & (* check if 1st character is '_' ASCII 95 *)
    (ASCII(rcv_other    '' , 2)) = 10 & (* check if 2nd character is 'LF' ASCII 10 *)
    (ASCII(rcv_other    '' , 18)) = 13 & (* check if 18th character is 'CR' ASCII 13 *)
    (ASCII(rcv_other    '' , 19)) = 10 then (* check if 19th character is 'LF' ASCII 10 *)

    lead_char_POS FALSE  := true; (* set POS data good flag *)
        (*extract 5 characters starting at 6th position*)
        rcv_char '481.2'  := (MID(rcv_other    '' ,5,6));
        rcv_number 481.2  := Any_to_Real (rcv_char '481.2' ); (* convert to a number *)
        rcv_real -481.2  := rcv_number 481.2 ; (* number is positive *)
        // rcvstring := ''; (* Clear input buffer *)
        rcvstring     ''  := DELETE (rcvstring     '' , 19, 1);
        rcv_other     ''  := DELETE (rcv_other     '' , 19, 1);
else
    lead_char_POS FALSE  := false; (* reset POS data good flag *)
    rcv_error_POS     ''  := rcvstring    '' ;
    rcv_length_ERR_POS    19  := rcv_string_length    19 ;
end_if;
```

Determine if there are errors in the ASCII string



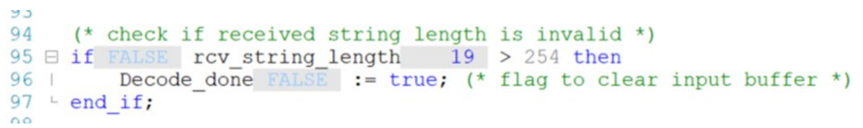
```
(* check for valid character string based on leading character *)
if FALSE (lead_char_NEG TRUE = 0) AND (lead_char_POS FALSE = 0) then
   Data_Error FALSE := true; (* set flag to signal error *)
   rcv_real -481.2 := 0; (* zero out invalid data *)
else
   Data_Error FALSE := false; (* reset error flag *)
end_if;
```

Send scale weight value to PLC memory



```
(* send scale weight data to PLC *)
Scale_wt_real -4812 := rcv_real -481.2 * 10;
Scale_wt_Intx10 -4812 := any_to_int(Scale_wt_real -4812 );
```

Indicate if string length is over running

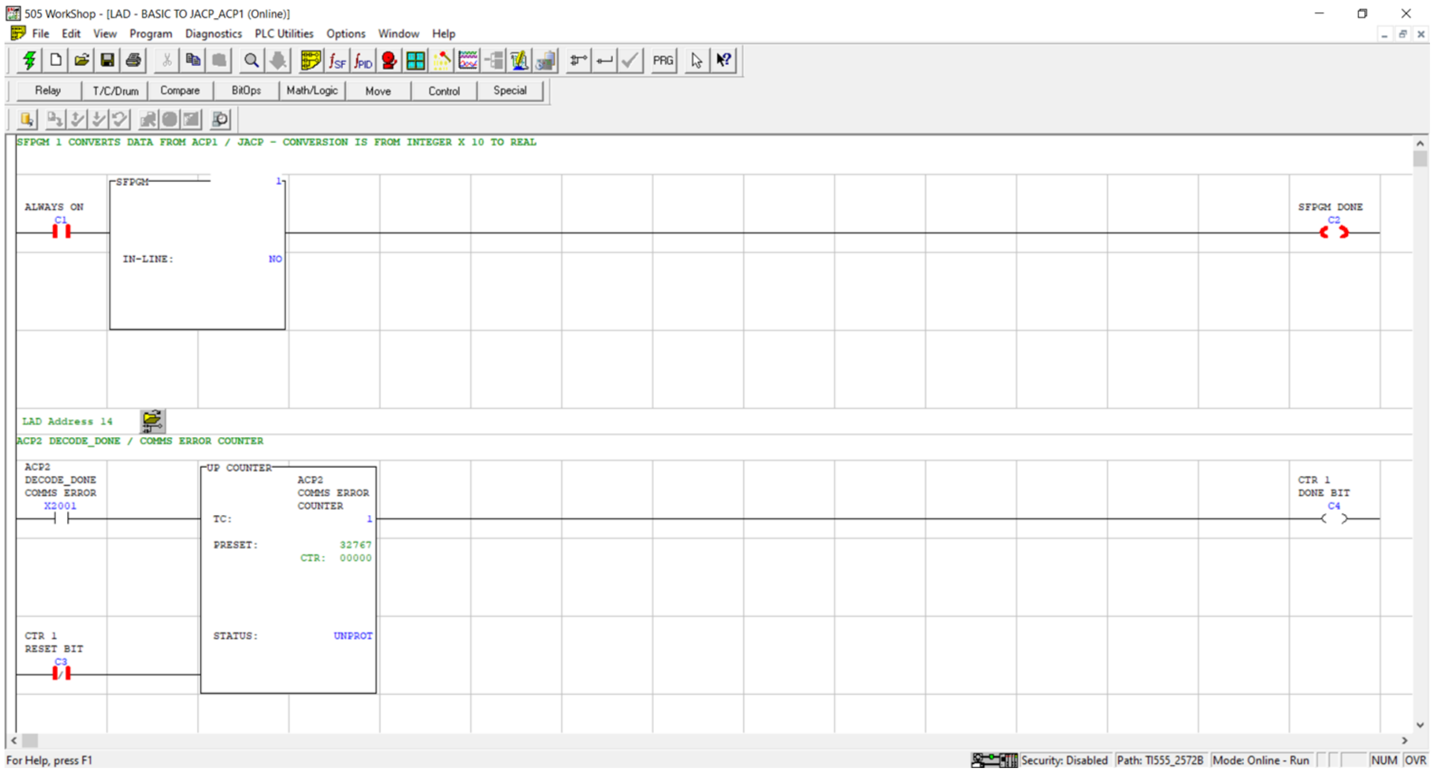


```
(* check if received string length is invalid *)
if FALSE rcv_string_length 19 > 254 then
    Decode_done FALSE := true; (* flag to clear input buffer *)
end_if;
```

Data received at PLC

In this Ladder Logic screen shot you can see where a Special Function Program is being called which is where the weight value is being handled.

Next you can see that X2001 is being used to count (or track) any decoding errors of the ASCII message.



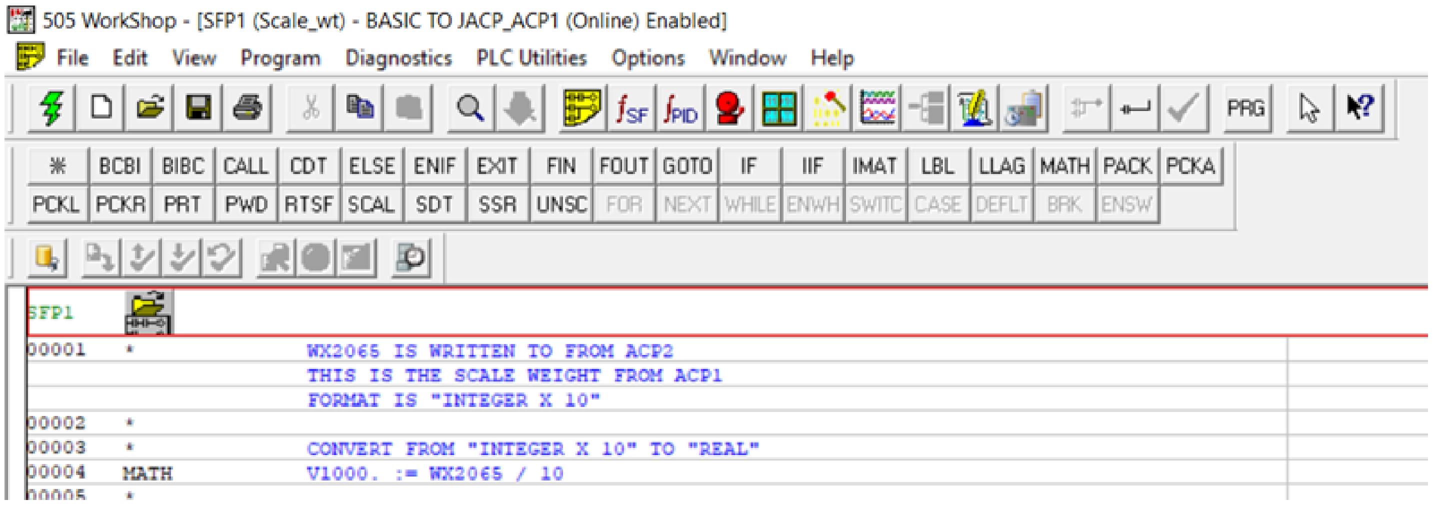
SFPGM 1 CONVERTS DATA FROM ACP1 / JACP - CONVERSION IS FROM INTEGER X 10 TO REAL

ACP2 DECODE_DONE / COMMS ERROR COUNTER

For Help, press F1 | Security: Disabled | Path: TI555_2572B | Mode: Online - Run | NUM | OVR

Data received at PLC

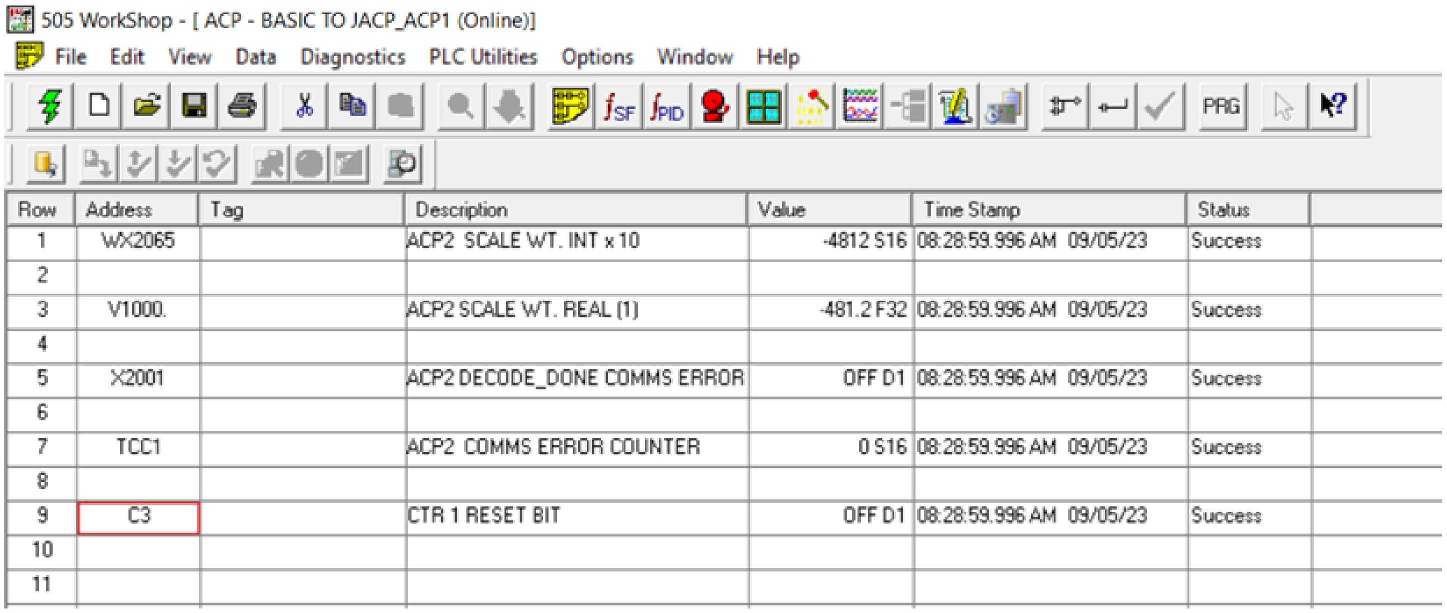
In the Special Function Program you can see where the ‘Integer x 10’ value from the ACP is being converted into a Real number.


505 WorkShop - [SFP1 (Scale_wt) - BASIC TO JACP_ACP1 (Online) Enabled]
File Edit View Program Diagnostics PLC Utilities Options Window Help
BCBI BIBC CALL CDT ELSE ENIF EXIT FIN FOUT GOTO IF IIF IMAT LBL LLAG MATH PACK PCKA
PCKL PCKR PRT PWD RTSF SCAL SDT SSR UNSC FOR NEXT WHILE ENWH SWITC CASE DEFLT BRK ENSW

```
SFP1
00001  *          WX2065 IS WRITTEN TO FROM ACP2
                  THIS IS THE SCALE WEIGHT FROM ACP1
                  FORMAT IS "INTEGER X 10"
00002  *
00003  *          CONVERT FROM "INTEGER X 10" TO "REAL"
00004  MATH       V1000. := WX2065 / 10
00005  *
```

In this Data Window you can see the values of the Scale Weight -

WX2065 shows up as ‘-4812’ which is in the format of ‘Integer x 10’
V1000. shows up as ‘-481.2’ which is in the format of ‘Real’ or ‘Floating Point’
X2001 which is the Boolean ‘Decode_done’ error bit
TCC1 which is the Counter tracking decode errors in ACP2


505 WorkShop - [ACP - BASIC TO JACP_ACP1 (Online)]
File Edit View Data Diagnostics PLC Utilities Options Window Help

Row	Address	Tag	Description	Value	Time Stamp	Status
1	WX2065		ACP2 SCALE WT. INT x 10	-4812 S16	08:28:59.996 AM 09/05/23	Success
2						
3	V1000.		ACP2 SCALE WT. REAL (1)	-481.2 F32	08:28:59.996 AM 09/05/23	Success
4						
5	X2001		ACP2 DECODE_DONE COMMS ERROR	OFF D1	08:28:59.996 AM 09/05/23	Success
6						
7	TCC1		ACP2 COMMS ERROR COUNTER	0 S16	08:28:59.996 AM 09/05/23	Success
8						
9	C3		CTR 1 RESET BIT	OFF D1	08:28:59.996 AM 09/05/23	Success
10						
11						

Appendix

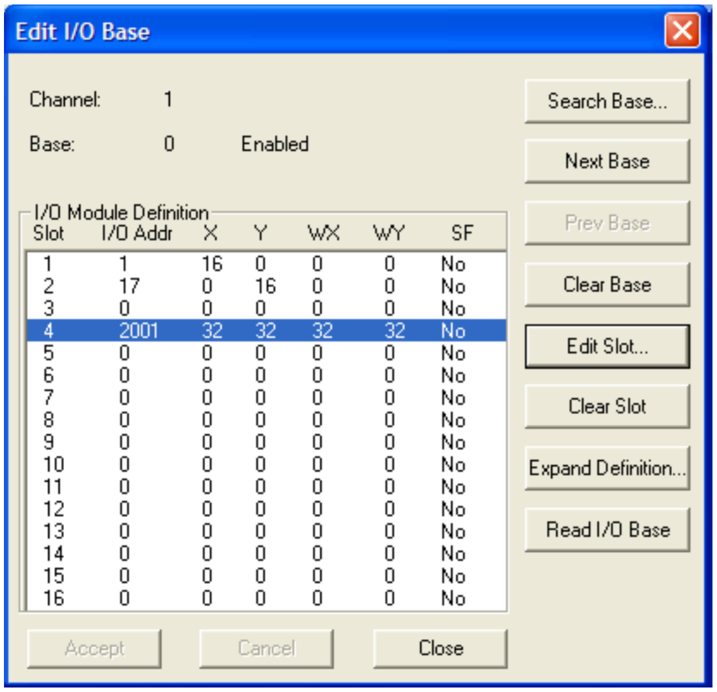
The ACP I/O interface is how the module reads & writes PLC memory.
In this example the I/O interface will be configured for 32 discrete points in, 32 discrete points out, 32 words in, & 32 words out.

2500P-ACP1 I/O Definition

The ACP1 I/O definition is specified by selecting one the following I/O configurations in the *CTI 2500P-ACP1 I/O*

Configuration Wizard provided in ***CTI Workbench***:

- Discrete I/O: 32 inputs / 32 outputs (32X/32Y)
- Word I/O: 32 inputs / 32 outputs (32WX/32WY)
- Mixed I/O: 32 discrete inputs/outputs and 32 word inputs/outputs (32X/32Y/32WX/32WY)



Channel: 1

Base: 0 Enabled

I/O Module Definition

Slot	I/O Addr	X	Y	WX	WY	SF
1	1	16	0	0	0	No
2	17	0	16	0	0	No
3	0	0	0	0	0	No
4	2001	32	32	32	32	No
5	0	0	0	0	0	No
6	0	0	0	0	0	No
7	0	0	0	0	0	No
8	0	0	0	0	0	No
9	0	0	0	0	0	No
10	0	0	0	0	0	No
11	0	0	0	0	0	No
12	0	0	0	0	0	No
13	0	0	0	0	0	No
14	0	0	0	0	0	No
15	0	0	0	0	0	No
16	0	0	0	0	0	No

The Mixed I/O interface requires special care when assigning an *I/O Address* because the Series 505® model allows one "login" address for each module slot. Therefore, the *I/O Address* assigns the image register positions for both the Discrete I/O and Word I/O values.
In the example, a value of "2001" is designated as the I/O Address. This equates to the following I/O mapping for ACP1 data within the PLC:

- 32 discrete inputs mapped to X2001-X2032
- 32 discrete outputs mapped to Y2033-Y2064
- 32 word inputs mapped to WX2065-WX2096
- 32 word outputs mapped to WY2097-WY2124

2500P-ACP1 I/O Configuration

From within WorkBench you open this configuration wizard and define:

Module Log-in Configuration—in this case it is 32X / 32Y / 32WX / 32WY

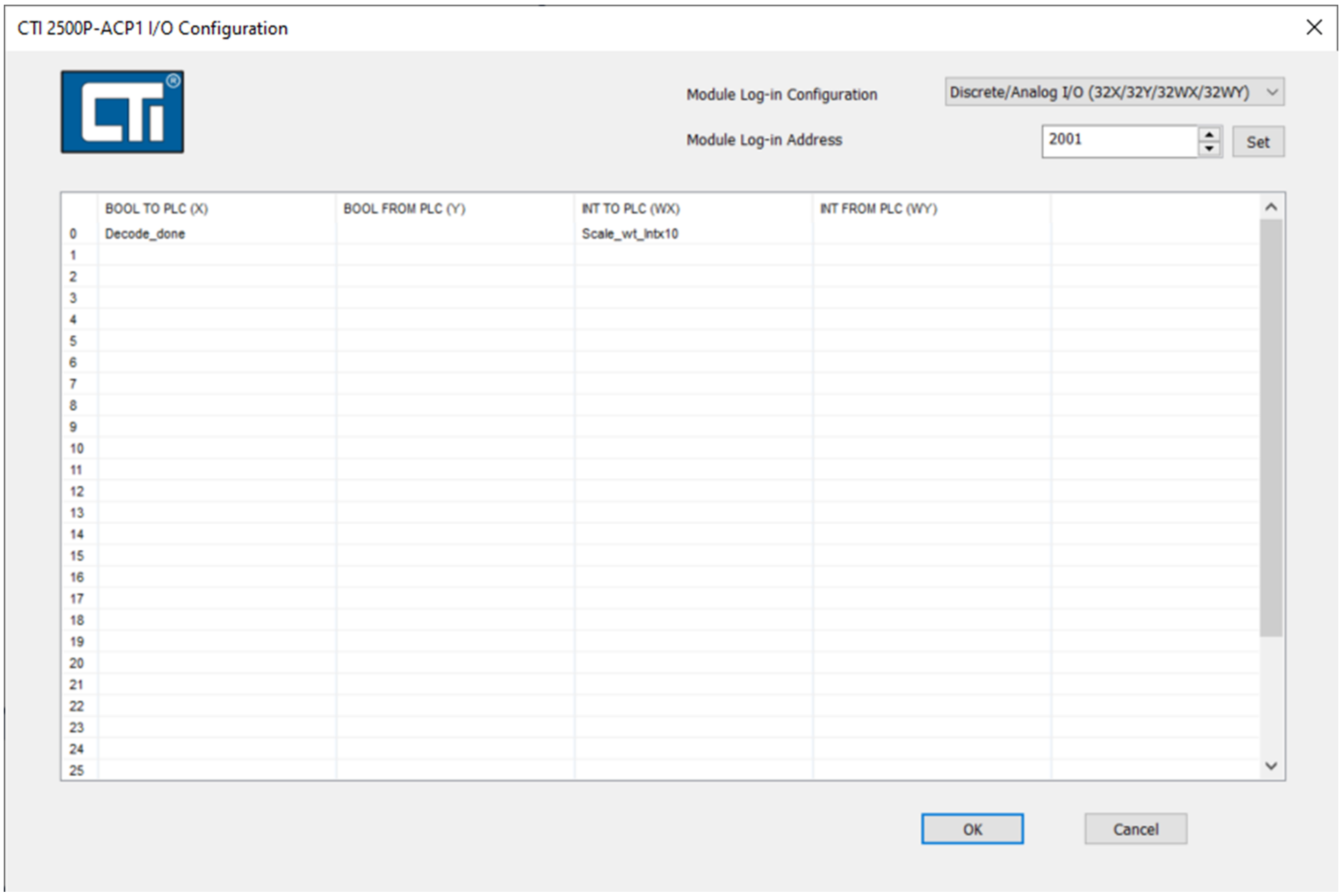
Module Log-In Address—this is the beginning I/O address of this configuration

Lastly, you populate the rows in each I/O column with the WorkBench symbol that you want to connect to PLC memory

In this example -

Boolean symbol 'Decode_done' has been assigned to X2001

Integer symbol 'Scale_wt_Intx10'has been assigned to WX2065



CTI 2500P-ACP1 I/O Configuration

Module Log-in Configuration: Discrete/Analog I/O (32X/32Y/32WX/32WY)

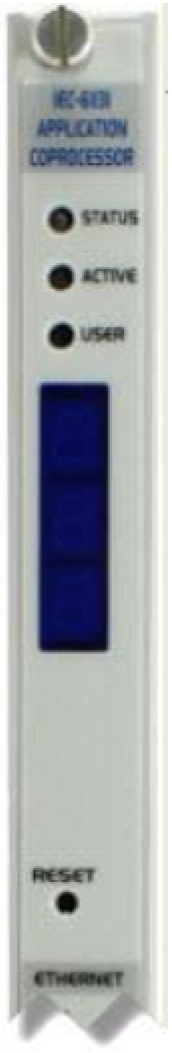
Module Log-in Address: 2001

	BOOL TO PLC (X)	BOOL FROM PLC (Y)	INT TO PLC (WX)	INT FROM PLC (WY)
0	Decode_done		Scale_wt_Intx10	
1				
2				
3				
4				
5				
6				
7				
8				
9				
10				
11				
12				
13				
14				
15				
16				
17				
18				
19				
20				
21				
22				
23				
24				
25				

2.1 Status Indicator LEDs

At the top of the module front panel are three status LEDs. The function of the LEDs is described in the following table.

LED	State	Indication
STATUS	Off	Module not operational
	Flashing	Module not ready – Operator action required (Module error, Watchdog timeout, Application program not found, or Host interface failure)
	On	Module operation is normal
ACTIVE	Off	Application program stopped
	Flashing	Program loaded but logic is not running (PAUSED state). I/O interface and communication protocols are active.
	On	Application program is executing (RUN state)
USER	Off	Controlled by application logic
	Flashing	
	On	



2.2 LED Multi-Segment Display

The Multi-Segment Display (MSD) is located below the status LEDs. The MSD is used to display status and error codes. During normal operation the MSD displays the TCP/IP address of the product, one octet at a time. When an error is encountered, the MSD will also display an Error Code. See *APPENDIX A: ERROR CODES* for a list of error codes and descriptions.

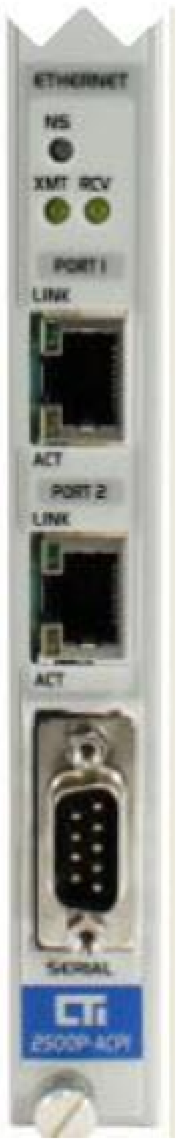
2.3 Reset Button

The Reset Button allows you to initiate a “soft reset” for the 2500P-ACP1 module. This reset is equivalent to cycling power to the module. When the button is depressed (using a pointed object such as a ball point pen, the module is restarted after an orderly shutdown of the application program. This reset action can be disabled by setting module switch (SW3) to CLOSED position. Section 3.1.2 contains information on location and setting for module switches.

2.4 Ethernet Status Indicators

The Ethernet LEDs indicate the state of the TCP/IP interface and whether the module is transmitting and receiving data via the Ethernet as shown in the following table.

LED	State	Indication
NS (Network Status)	Off	TCP/IP is not operational.
	On-Red	TCP/IP is operational. A device with the same IP address as this 2500P-ACP1 module has been detected.
	On-Green	TCP/IP is connected and operational.
XMT (Transmit)	Flashing	Ethernet port is transmitting data
RCV (Receive)	Flashing	Ethernet port is receiving data.



2.5 Ethernet Ports

The 2500P-ACP1 provides two Ethernet ports capable of operating at 10/100Mb, half or full duplex. The speed and duplex mode are automatically negotiated with the device connected to the port. Each port supports auto-crossover capability, allowing the port to be connected to an external Ethernet switch or directly to a device, such as a laptop or 2500 Series® controller. Both ports are functionally equivalent.

The 2500P-ACP1 incorporates an Ethernet switch which is connected to both Ethernet ports and the microprocessor. The switch allows either port to communicate with the microprocessor. The two ports can be connected or isolated from each other (see 'Port Isolation' below). Besides providing this connectivity, the switch also provides hardware protection against network broadcast/multicast storms.

It is also possible to enable 'IP aliasing" by configuring an Alias IP Address and Alias Subnet Mask. This allows two IP addresses to be associated with the ACP1 module so that each Ethernet port can be connected to a separate sub-network. When 'IP aliasing' is enabled, either Ethernet port can be used with either sub-network (i.e. the IP Address and Subnet Mask is not port specific). When using this feature, Port Isolation should always be enabled.

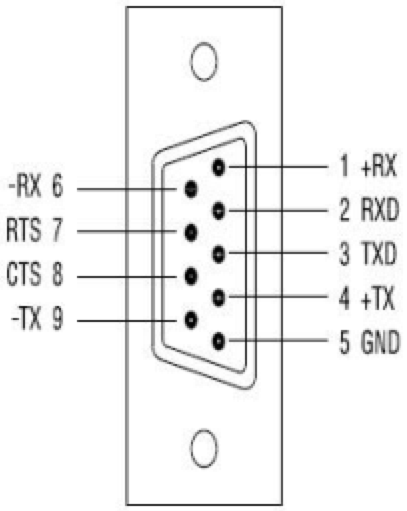
Port Isolation can be enabled by setting module switch (SW4) to CLOSED position. This setting blocks forwarding of all Ethernet packets between the two ports and allows the ACP1 to be used with redundant network topologies without creating network loops. Section 3.1.2 contains information on location and setting for module switches.

Each Ethernet port connector contains two embedded LEDs. The LINK LED indicates whether the Ethernet port is successfully connected to another Ethernet device, such as a network switch. The ACTIVITY (ACT) LED provides visual indication that Ethernet packets are being received or transmitted via the port. See the following table below for more information.

LED	State	Indication
Link	Off	Ethernet link is not available.
	On	Ethernet link is available.
Act (Activity)	Off	No Ethernet frames are being transmitted on the network to which the port is connected.
	Flashing	Ethernet frames are being transmitted on the network to which the port is connected

2.7 Serial Port

The male DB9 connector on the front panel provides the serial port interface. Modbus-RTU (Master or Slave) and General ASCII Send/Receive data protocols are supported for the serial port and managed by the application program. All port parameters, including the selected electrical interface (RS-232 or RS-422), are set by software configuration via ***CTI Workbench***. The cable used with the external device must connect to the pins used by the selected electrical interface.



RS-232 (Subset) Pinout:

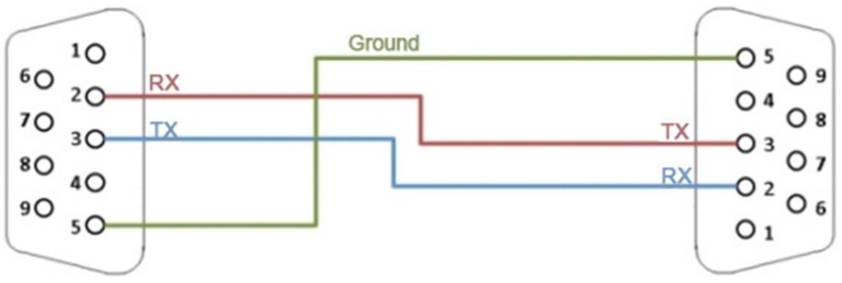
Pin	Signal	Description
2	RXD	Receive Data
3	TXD	Transmit Data
5	GND	Signal Ground
7	RTS	Request to Send (optional)
8	CTS	Clear to Send (optional)

RS-422 Pinout:

Pin	Signal	Description
1	+RX	Receive Data (+)
4	+TX	Transmit Data (+)
6	-RX	Receive Data (-)
9	-TX	Transmit Data (-)

NOTE:
A serial port connection to CTI Workbench is not supported by the ACP1 module.
This interface must be made using TCP/IP connection.

Null Modem cable pinout:



Serial Port Communications

CTI products such as the 2500P-ACP1 and 2500P-JACP provide a serial port which can be used to communicate with devices support electrical interfaces such as RS-232, RS-422, and RS 485. There are two requirements for communication with the device..

1. The port parameters must be configured to match the requirements of the target device.
2. A communications protocol that is supported by the device must be used.

Configuring the Serial Port Parameters.

The serial port is configured by constructing an ASCII string containing parameter descriptors and associated values as shown in the following table.

Parameter	Descriptor	Valid Values	Default Value
Port ID	PT	*Product Dependent*	1
Baud Rate	BD	1200,2400, 4800, 9600, 19200, 38400, 57600, 115200	9600
Data Bits	DB	7, 8	8
Stop Bits	SB	1,2	1
Parity	PY	None (N), Even (E), Odd (O)	ASCII (N) / Modbus RTU (E)
Flow Control	FC	No (N), Yes (Y) *"Y"enables RTS-CTS handshake (CTS must beTRUE to send)*	N
Interface	IF	RS232, RS422 (ACP1 and JACP modules), RS-485 (JACP Module only)	RS232

Usage Rules

- If any parameters are missing or assigned invalid values, the default value for the parameter(s) will be used.
- All characters in the string are case insensitive.
- The string is not order dependent.
- Any extraneous content included in the string will be ignored.

Example String

PT=1 BD=19200 DB=8 SB=1 PY=N FC=N IF=RS232

Choosing a Communications Protocol

The following communications protocol are available

- Modbus RTU Master Fieldbus configuration for "Com port:"
- Modbus RTU Slave Function Blocks: MBSLAVERTU, MBSLAVERTUEX
- General ASCII; SERIO and SERIO_B to send and receive character string or byte array.

CTI Product Support

CTI 2500P-ACP1
CTI JACP Module

ACP1 Functions and Instructions used in this application

SERIO

Function Block - Manages communications through the serial port

Inputs

RUN : BOOL	Enables communications
SND : BOOL	*TRUE* Sends data
CONF : STRING	Contains Serial Port Parameters
DATASND : STRING	Contains the data to be sent

Outputs

OPEN : BOOL	*TRUE* if the communication port is open
RCV : BOOL	*TRUE* if data has been received
ERR : BOOL	*TRUE* if error detected during sending data
DATARCV : STRING	Contains received data

Remarks

The **RUN** input does not include an edge detection. The block tries to open the port on each call when **RUN** is *TRUE* (if port is not already open).
When **RUN** is *FALSE* the port will be closed (if open).
The **CONF** input is used for settings when opening the port. See *Serial Port Parameters*.
The **SND** input does not include an edge detection. Characters are sent on each call if **SND** is *TRUE* and **DATASND** is not empty.
The **DATARCV** string is erased and replaced with any received data each cycle.
Your application is responsible for storing received character immediately after each call to SERIO block.

The SERIO function block can be used in PC simulation mode.
In that case, the **CONF** input defines the communication port according to the syntax of the *MODE* command.
For example:

```
     'COM1:9600,N,8,1'
```

ST Language

MySer is a declared instance of SERIO function block.

```
MySer (RUN, SND, CONF, DATASND);
OPEN := MySer.OPEN;
RCV := MySer.RCV;
ERR := MySer.ERR;
DATARCV := MySer.DATARCV;
```

String Operations

Below are the standard operators and functions that manage character strings:

Code	Operator / Function
+	concatenation of strings
CONCAT	concatenation of strings
MLEN	get string length
DELETE	delete characters in a string
INSERT	insert characters in a string
FIND	find characters in a string
REPLACE	replace characters in a string
LEFT	extract a part of a string on the left
RIGHT	extract a part of a string on the right
MID	extract a part of a string
CHAR	build a single character string
ASCII	get the ASCII code of a character within a string
ATOH	converts string to integer using hexadecimal basis
HTOA	converts integer to string using hexadecimal basis
CRC16	CRC16 calculation
ArrayToString	copies elements of an **SINT** array to a **STRING**
StringToArray	copies characters of a **STRING** to an **SINT** array

Other functions are available for managing string tables as resources:

Function	Description
StringTable	Select the active string table resource
LoadString	Load a string from the active string table

Constant Expressions

Constant expressions can be used in all languages for assigning a variable with a value. All constant expressions have a well defined data type according to their semantics. If you program an operation between variables and constant expressions having inconsistent data types, it will lead to syntax errors when the program is compiled. Below are the syntax rules for constant expressions according to possible data types:

BOOL: Boolean

There are only two possible Boolean constant expressions. They are reserved keywords *TRUE* and *FALSE*.

SINT: Small (8 bit) Integer

Small integer constant expressions are valid integer values (between -128 and 127) and must be prefixed with *SINT#*. All integer expressions having no prefix are considered as **DINT**integers.

USINT / BYTE: Unsigned 8 bit Integer

Unsigned small integer constant expressions are valid integer values (between 0 and 255) and must be prefixed with *USINT#*. All integer expressions having no prefix are considered as **DINT**integers.

INT: 16 bit Integer

16 bit integer constant expressions are valid integer values (between -32768 and 32767) and must be prefixed with *INT#*.
All integer expressions having no prefix are considered as *DINT* integers.

UINT / WORD: Unsigned 16 bit Integer

Unsigned 16 bit integer constant expressions are valid integer values (between 0 and 65535) and must be prefixed with *UINT#*.
All integer expressions having no prefix are considered as *DINT* integers.

DINT: 32 bit (default) Integer

32 bit integer constant expressions must be valid numbers between -2147483648 to +2147483647. DINT is the default size for integers: such constant expressions do not need any prefix. You can use *2#*, *8#* or *16#* prefixes for specifying a number in respectively binary, octal or hexadecimal basis.

UDINT / DWORD: Unsigned 32 bit Integer

Unsigned 32 bit integer constant expressions are valid integer values (between 0 and 4294967295) and must be prefixed with *UDINT#*.
All integer expressions having no prefix are considered as **DINT** integers.

LINT: Long (64 bit) Integer

Long integer constant expressions are valid integer values and must be prefixed with *LINT#*.
All integer expressions having no prefix are considered as **DINT** integers.

REAL: Single precision Floating Point Value

Real constant expressions must be a valid number, and must include a decimal point ("."). If you need to enter a real expression having an integer value, add *.0* at the end of the number.
You can use *F* or *E* separators for specifying the exponent when entering a value using scientific notation.
REAL is the default precision for floating point numbers. Such expressions do not need any prefix.

LREAL: Double Precision Floating Point Value

Real constant expressions must be valid number, must include a decimal point ("."), and must be prefixed with *LREAL#*.
If you need to enter a real expression having an integer value, add *.0* at the end of the number.
You can use *F* or *E* separators for specifying the exponent when entering a value using scientific notation.

TIME: Time

Time constant expressions can be used to represent durations of less than 24 hours. Expressions must be prefixed by either *TIME#* or *T#*.
They are expressed as a number of hours followed by *h*, a number of minutes followed by *m*, a number of seconds followed by *s*, and a number of milliseconds followed by *ms*.
The order of units (hour, minutes, seconds, milliseconds) must be respected. You cannot insert blank characters within the time expression.
There must be at least one valid time unit letter in the expression. See examples below:

- Declare a variable cycletime with data type TIME. Following examples are valid:

```
cycletime := t#12s; //Sets cycletime to 12 seconds
cycletime :=time#1m100ms; //Sets cycletime to 1 minute plus 100 milliseconds
cycletime := t#1h10m5s50ms; //Sets cycletime to 1 hour, 10 minutes, 5 seconds, and 50 milliseconds.
```

STRING: Character String

String expressions must be written between single quote marks. The length of the string cannot exceed 255 characters.
You can use the following sequences to represent a special or not printable character within a string:

Sequence	Description
$$	"$" character
$'	Single quote
$T	Tab stop (ASCII code 9)
$R	Carriage return character (ASCII code 13)
$L	Line feed character (ASCII code 10)
$N	Carriage return plus line feed characters (ASCII codes 13 and 10)
$P	Page break character (ASCII code 12)
$xx	Any character (xx is the ASCII code expressed as two hexadecimal digits)

Example

Below are some examples of valid constant expressions:

Expression	Description
TRUE	*TRUE* boolean expression
FALSE	*FALSE* boolean expression
SINT#127	Short integer
INT#2000	Signed 16-bit integer
123456	**DINT** (32 bit) integer
16#abcd	**DINT** integer in hexadecimal basis
LINT#1	Long (64 bit) integer having the value "1"
0.0	0 expressed as a **REAL**number
1.002E3	1002 expressed as a **REAL**number in scientific notation
LREAL#1E-200	Double precision real number
T#23h59m59s999ms	Maximum **TIME**value
TIME#0s	Null **TIME**value
T#1h123ms	**TIME**value with some units missing
'hello'	Character string
'name$Tage'	Character string with two words separated by a tab
'I$'m here'	Character string with a quote inside (I'm here)
'x$00y'	Character string with two characters separated by a null character (ASCII code 0)

Sequence	Description
$$	"$" character
$'	Single quote
$T	Tab stop (ASCII code 9)
$R	Carriage return character (ASCII code 13)
$L	Line feed character (ASCII code 10)
$N	Carriage return plus line feed characters (ASCII codes 13 and 10)
$P	Page break character (ASCII code 12)
$xx	Any character (xx is the ASCII code expressed as two hexadecimal digits)

Example

Below are some examples of valid constant expressions:

Expression	Description
TRUE	*TRUE* boolean expression
FALSE	*FALSE* boolean expression
SINT#127	Short integer
INT#2000	Signed 16-bit integer
123456	**DINT** (32 bit) integer
16#abcd	**DINT** integer in hexadecimal basis
LINT#1	Long (64 bit) integer having the value "1"
0.0	0 expressed as a **REAL**number
1.002E3	1002 expressed as a **REAL**number in scientific notation
LREAL#1E-200	Double precision real number
T#23h59m59s999ms	Maximum **TIME**value
TIME#0s	Null **TIME**value
T#1h123ms	**TIME**value with some units missing
'hello'	Character string
'name$Tage'	Character string with two words separated by a tab
'I$'m here'	Character string with a quote inside (I'm here)
'x$00y'	Character string with two characters separated by a null character (ASCII code 0)

Below are some examples of typical errors in constant expressions:

Expression	Error-Description
BoolVar := 1;	0 and 1 cannot be used for Booleans - must use *TRUE* or *FALSE*
1a2b	Base prefix ("16#") omitted
1E-200	"LREAL#" prefix omitted for a double precision float
T#12	Time unit missing
'I'm here'	Quote within a string with "$" mark omitted
hello	Quotes omitted around a character string

IF THEN ELSE ELSIF END_IF

Statement - Conditional execution of statements

Syntax

```
    IF <BOOL expression> THEN
        <statements>
    ELSIF <BOOL expression> THEN
        <statements>
    ELSE
        <statements>
    END_IF;
```

Remarks

The `IF` statement is available in ST only.
The execution of the statements is conditioned by a Boolean expression.
`ELSIF` and `ELSE` statements are optional.
Multiple `ELSIF` statements can be used when desired.

ST Language

```
(* simple condition *)
IF bCond THEN
   Q1 := IN1;
   Q2 := TRUE;
END_IF;

(* binary selection *)
IF bCond THEN
   Q1 := IN1;
   Q2 := TRUE;
ELSE
   Q1 := IN2;
   Q2 := FALSE;
END_IF;

(* enumerated conditions *)
IF bCond1 THEN
   Q1 := IN1;
ELSIF bCond2 THEN
   Q1 := IN2;
ELSIF bCond3 THEN
   Q1 := IN3;
ELSE
   Q1 := IN4;
END_IF;
```

CONCAT

Function - Concatenate strings

Inputs

IN_1 : STRING Any string variable or constant expression
...
IN_N : STRING Any string variable or constant expression

Outputs

Q : STRING   Concatenation of all inputs

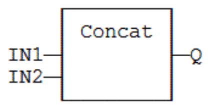
Remarks

In FBD or LD language, the block may have up to 16 inputs.
In LD language,the input (EN) enables the operation, and the output (ENO) keeps the same value as the input.
In IL or ST, the function accepts a variable number of inputs (at least 2).
Note that you also can use the "+" operator to concatenate strings.

ST Language

```
Q := CONCAT ('AB', 'CD', 'E');
(* now Q is 'ABCDE' *)
```

FBD Language



MLEN

Function - Get the number of characters in a string

Inputs

IN : STRING Character string

Outputs

NBC : DINT Number of characters currently in the string (0 if string is empty)

Remarks

In LD language, the input (EN) enables the operation, and the output (ENO) keeps the same value as the input.
In IL, the first input (IN: STRING) must be loaded on the stack before calling the function.

ST Language

```
NBC := MLEN (IN);
```

ASCII

Function - Get the ASCII code of a character within a string

Inputs

IN : STRING Input string
POS : DINT Position of the character within the string
(the first valid position is 1).

Outputs

CODE : DINT ASCII code of the selected character.
(or 0 if position is invalid)

Remarks

In LD language, the input (EN) enables the operation, and the output (ENO) keeps the same value as the input.
In IL language, the first parameter (IN) must be loaded on the stack before calling the function. The other input is the operand of the function.

ST Language

```
CODE := ASCII (IN, POS);
```

MID

Function - Extract characters of a string starting at any position within the string

Inputs

IN : STRING Character string
NBC : DINT Number of characters to extract
POS : DINT Position of the first character to extract (first character of IN is at position 1).

Outputs

Q : STRING String containing the first NBC characters of IN.

Remarks

The first valid position in the string is 1.
The number of characters extracted is limited to smallest value of: IN string length, Q string length, or (POS + NDEL-1).
In LD language, the input (EN) enables the operation, and the output (ENO) keeps the same value as the input.
In IL, the first input (IN: STRING) must be loaded on the stack before calling the function. Other argument are operands of the function, separated by commas.

ST Language

```
Q := MID (IN, NBC, POS);
```

ANY_TO_REAL

Operator - Converts the input into a single-precision Real value

Inputs

IN : ANY Input value

Outputs

Q : REAL Value converted to 32-bit Real number

Remarks

For **BOOL** input data types, the output is 0.0 or 1.0.
For any **INTEGER** input data type, the output is a **REAL** number with the same value.
For **TIME** input data types, the result is the number of milliseconds.
For **STRING** inputs, the output is the number represented by the string, or 0.0 if the string does not represent a valid number.
In LD language, the operation executes only if the input (EN) is *TRUE*. The output (ENO) keeps the same value as the input.
In IL Language, the **ANY_TO_REAL** function converts the value pushed on the stack.

ST Language

```
Q := ANY_TO_REAL (IN);
```

NEG -

Operator - Performs an integer negation of the input

Inputs

IN : DINT Integer value

Outputs

Q : DINT Integer negation of the input

Truth table (examples)

IN	Q
0	0
1	-1
-123	123

Remarks

In FBD and LD language, the block NEG can be used.
In LD language, the operation executes only if the input (EN) is *TRUE*. The output (ENO) keeps the same value as the input.
This feature is not available in IL language.
In ST language, "-" can be followed by a complex Boolean expression between parenthesis.

ST Language

```
Q := -IN;
Q := - (IN1 + IN2);
```

DELETE

Function - Delete characters in a string

Inputs

IN : STRING Character string
NBC : DINT Number of characters to be deleted
POS : DINT Position of the first deleted character (first character position is 1)

Outputs

Q : STRING Modified string.

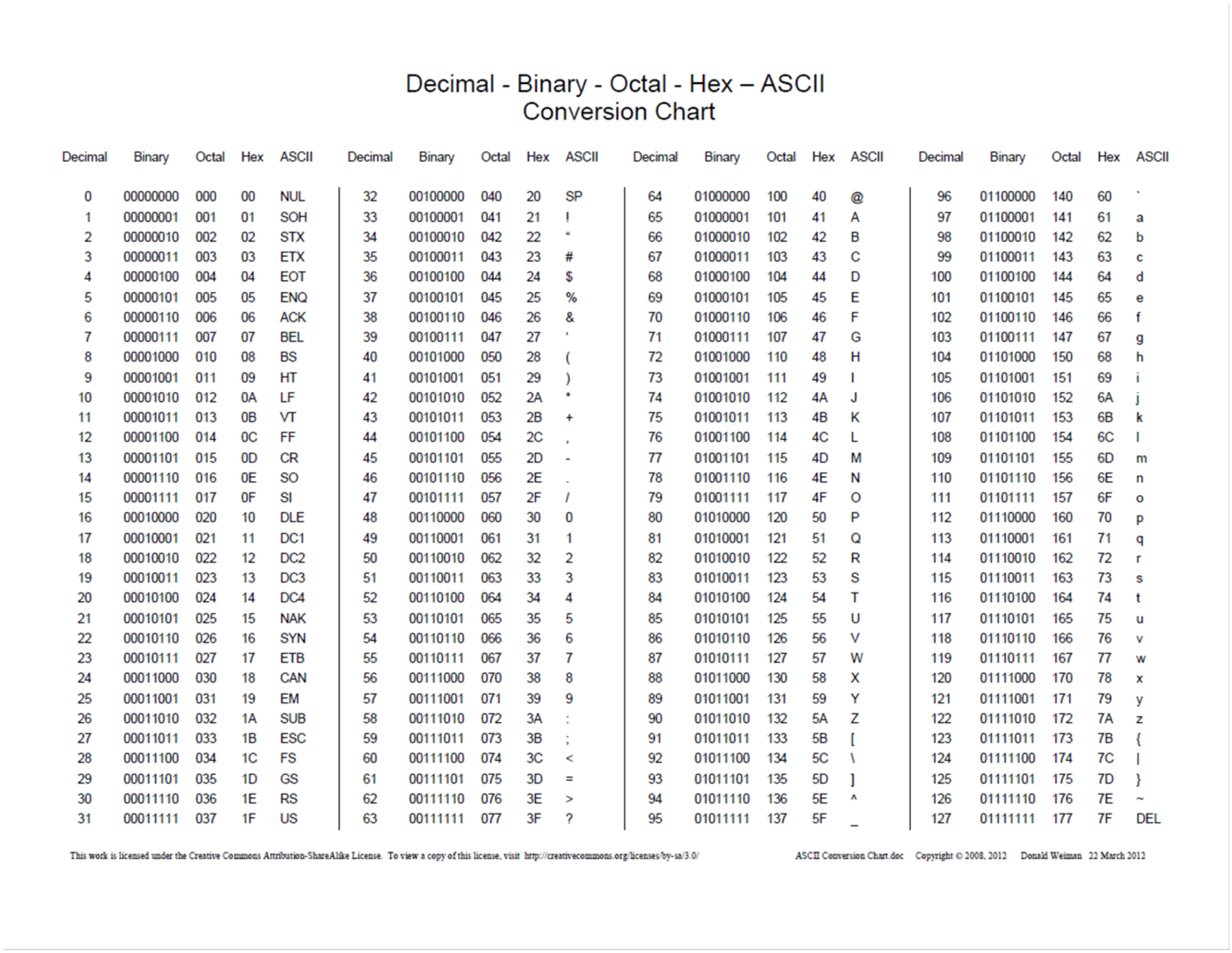
Remarks

The first valid character position is 1.
In LD language, the input (EN) enables the operation, and the output (ENO) keeps the same value as the input.
In IL, the first input (IN: STRING) must be loaded on the stack before calling the function. Other arguments are operands of the function, separated by commas.

ST Language

```
Q := DELETE (IN, NBC, POS);
```

ASCII conversion chart



Decimal - Binary - Octal - Hex – ASCII
Conversion Chart

Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII	Decimal	Binary	Octal	Hex	ASCII
0	00000000	000	00	NUL	32	00100000	040	20	SP	64	01000000	100	40	@	96	01100000	140	60	`
1	00000001	001	01	SOH	33	00100001	041	21	!	65	01000001	101	41	A	97	01100001	141	61	a
2	00000010	002	02	STX	34	00100010	042	22	"	66	01000010	102	42	B	98	01100010	142	62	b
3	00000011	003	03	ETX	35	00100011	043	23	#	67	01000011	103	43	C	99	01100011	143	63	c
4	00000100	004	04	EOT	36	00100100	044	24	$	68	01000100	104	44	D	100	01100100	144	64	d
5	00000101	005	05	ENQ	37	00100101	045	25	%	69	01000101	105	45	E	101	01100101	145	65	e
6	00000110	006	06	ACK	38	00100110	046	26	&	70	01000110	106	46	F	102	01100110	146	66	f
7	00000111	007	07	BEL	39	00100111	047	27	'	71	01000111	107	47	G	103	01100111	147	67	g
8	00001000	010	08	BS	40	00101000	050	28	(	72	01001000	110	48	H	104	01101000	150	68	h
9	00001001	011	09	HT	41	00101001	051	29	)	73	01001001	111	49	I	105	01101001	151	69	i
10	00001010	012	0A	LF	42	00101010	052	2A	*	74	01001010	112	4A	J	106	01101010	152	6A	j
11	00001011	013	0B	VT	43	00101011	053	2B	+	75	01001011	113	4B	K	107	01101011	153	6B	k
12	00001100	014	0C	FF	44	00101100	054	2C	,	76	01001100	114	4C	L	108	01101100	154	6C	l
13	00001101	015	0D	CR	45	00101101	055	2D	-	77	01001101	115	4D	M	109	01101101	155	6D	m
14	00001110	016	0E	SO	46	00101110	056	2E	.	78	01001110	116	4E	N	110	01101110	156	6E	n
15	00001111	017	0F	SI	47	00101111	057	2F	/	79	01001111	117	4F	O	111	01101111	157	6F	o
16	00010000	020	10	DLE	48	00110000	060	30	0	80	01010000	120	50	P	112	01110000	160	70	p
17	00010001	021	11	DC1	49	00110001	061	31	1	81	01010001	121	51	Q	113	01110001	161	71	q
18	00010010	022	12	DC2	50	00110010	062	32	2	82	01010010	122	52	R	114	01110010	162	72	r
19	00010011	023	13	DC3	51	00110011	063	33	3	83	01010011	123	53	S	115	01110011	163	73	s
20	00010100	024	14	DC4	52	00110100	064	34	4	84	01010100	124	54	T	116	01110100	164	74	t
21	00010101	025	15	NAK	53	00110101	065	35	5	85	01010101	125	55	U	117	01110101	165	75	u
22	00010110	026	16	SYN	54	00110110	066	36	6	86	01010110	126	56	V	118	01110110	166	76	v
23	00010111	027	17	ETB	55	00110111	067	37	7	87	01010111	127	57	W	119	01110111	167	77	w
24	00011000	030	18	CAN	56	00111000	070	38	8	88	01011000	130	58	X	120	01111000	170	78	x
25	00011001	031	19	EM	57	00111001	071	39	9	89	01011001	131	59	Y	121	01111001	171	79	y
26	00011010	032	1A	SUB	58	00111010	072	3A	:	90	01011010	132	5A	Z	122	01111010	172	7A	z
27	00011011	033	1B	ESC	59	00111011	073	3B	;	91	01011011	133	5B	[	123	01111011	173	7B	{
28	00011100	034	1C	FS	60	00111100	074	3C	<	92	01011100	134	5C	\	124	01111100	174	7C	|
29	00011101	035	1D	GS	61	00111101	075	3D	=	93	01011101	135	5D	]	125	01111101	175	7D	}
30	00011110	036	1E	RS	62	00111110	076	3E	>	94	01011110	136	5E	^	126	01111110	176	7E	~
31	00011111	037	1F	US	63	00111111	077	3F	?	95	01011111	137	5F	_	127	01111111	177	7F	DEL

10SEP2023

Control Technology Inc.
5734 Middlebrook Pike, Knoxville, TN 37921-5962
Phone: +1.865.584.0440   Fax: +1.865.584.5720
www.controltechnology.com

ROCK SOLID PERFORMANCE. TIMELESS COMPATIBILITY.