# ComGenius for SoftPLC® Runtime

Version 1.0

# Table of Contents

# Chapter 1. Overview

## 1.1. Introduction

This document describes the installation, usage, and functionality of a TLM (TOPDOC Loadable Module) for SoftPLC version 4.x and later. The TLM implements several **TLIs** (TOPDOC Loadable Instructions) for use with **serial ports**. COMGENIUS can be used to perform bi-directional serial communications to devices via RS-232, RS-422, or RS-485 links. COMGENIUS supports up to 36 serial ports in a SoftPLC system.

SoftPLC Corporation's SoftPLC control software product employs a unique technology which lets C/C++ or Java language programmers add new ladder logic instructions to the instruction set. A loadable instruction of this type is called a **TLI**. TOPDOC, the ladder logic programming package which supports SoftPLC, automatically learns about new TLIs as it logs into a SoftPLC.

TLIs may be developed by any competent C/C/Java programmer who has access to the SoftPLC C/C/Java Programmer's Toolkit, a product readily available from SoftPLC Corporation. There are a number of Systems Integrators who are SoftPLC Partners who possess the requisite expertise. End users may also have this capability.

This document describes a number of TLIs, all which reside in a **TLM** (TOPDOC Loadable Module) that is provided at no charge with all SoftPLC licenses. The TLM described by this document is called **COMGENIUS**. COMGENIUS can be used to perform bi-directional serial communications to devices via RS-232, RS-422, or RS-485 links. COMGENIUS supports up to 36 serial ports in a SoftPLC system.

## 1.2. Definitions

| | |
|---|---|
| **TLM** | is a TOPDOC Loadable Module that you write in C or C++. It can implement a driver and/or one or more ladder instructions. It is executable within SoftPLC only, so it must be downloaded. |
| **TLI** | A custom written ladder instruction, meaning TOPDOC Loadable Instruction. A TLI can be written in C/C++ and reside in a TLM, or it can be written in Java and reside in a Modlet or Driverlet. |
| **port** | is a serial port. The SoftPLC runtime and COMGENIUS both support up to 36 serial ports. |
| **byte** | is an 8 bit value. |
| **character** | is a 16 bit value that can reside in a STRING element and holds an international UCS-2 (UNICODE) symbol. ASCII is a subset of UCS-2. |

| | |
|---|---|
| **STRING element** | is an array of 16 bit (international) characters up to 82 character in length. The SoftPLC runtime supports arrays of STRING elements called STRING files, where there may be up to 10,000 STRING elements in each file. Up to 10,000 files are supported in a SoftPLC runtime. This is a maximum total of about 100,000,000 STRING elements comprised of 8,200,000,000 UCS-2 characters. |
| **format string** | is used by the COMPRINT and STRPRINT TLIs and describes how to how to convert the TLI's input arguments to a sequence of characters for output to a serial port or to a STRING element. |
| **whitespace** | a whitespace character is any character that is one of: ' ' (blank), \t (tab), \n (line feed), \r (carriage return), \v (vertical tab), and \f (formfeed). |

## 1.3. TLI Summary

The TLIs whose names begin with "COM" take a parameter called **port**. The port number specifies which logical serial COM channel a TLI is to work on. Since COMGENIUS supports 36 ports, legal values for port are 0 through 35. The port number may be provided either as a constant or as a PLC datatable word containing the value. All supported ports may be concurrently active.

The TLIs included in COMGENIUS TLM are:

| TLI Name | Description |
|---|---|
| COMPRINT | Outputs a packet of characters which is dynamically assembled using powerful string formatting. |
| COMSCAN | Receives a packet of bytes and automatically decomposes it into useful data fields located in INTEGER, FLOAT and/or STRING elements. |
| COMRCVCLEAR | Clears the input buffer for a port. |
| COMRCVSTS | Tells how many characters are in the input buffer for a port. |
| COMXMITCLEAR | Clears the output buffer for a port. |
| STRPRINT | Works like COMPRINT, but uses a datatable resident destination STRING element instead of using a port. Outputs a string of characters using powerful string formatting. |
| STRSCAN | Works like COMSCAN, but uses a datatable resident STRING element instead of characters in a port buffer. Automatically decomposes a STRING into useful data fields located in INTEGER, FLOAT and/or STRING elements. |

# 1.4. Requirements

- TOPDOC NexGen version 1.3.06xxxx or later, since there were some mandatory ehancements made in early 2006 which dealt with STRING parameters during instruction entry.

- Version 4.x SoftPLC or later.

- Serial Ports. CPU offerings from SoftPLC Corp. come standard with COM1 and sometimes COM2 depending on the platform. COM3-COM36 can only be used if you have one or more supported serial port expansion cards installed in the system. Check with tech support at SoftPLC Corp. for a list of supported serial port expansion cards available from SoftPLC for your platform.

# 1.5. Changes in 4.x

This TLM was substantially re-written since version 3.x SoftPLC, where it was known as **COMGENIE**. Below are the major changes from the 3.x version:

- All configuration information is now given in a single text file called COMGENIUS.LST. Information that was in the STRINGS.TXT file must be moved into COMGENIUS.LST. (This can be done with a simple cut and paste operation.)

- Internally, format strings are comprized of 16 bit characters. So although COMGENIUS.LST is an 8 bit ASCII file, any arbitrary UCS-2 character may be put into a format string by using the **unicode escape** sequence, which is like \uHHHH where the Hs are hex characters 0-9 and A-F or a-f. For example, the sequence \u000a is an ASCII line feed. Unicode escapes are an extension of the single byte **hex escape**, which have always been supported and still are, \xHH.

- The special format string escape \n is now interpreted as a line feed character, and **not** a carriage return linefeed combination as before. If you are converting from 3.x, you may have to do a search and replace on your format strings, changing \n with \r\n.

- The port configuration information is now also in the COMGENIUS.LST file and not onthe module command line as before.

- STRING datatable elements are now required for COMPRINT or COMSCAN instructions where the corresponding format string is using %s. This is easier to work with, but might require some logic changes since you may no longer print a string contained in an integer file.

- The MEMCOMPARE TLI has been dropped. For comparing integers and floats, EQU or CMP may be used. For comparing STRINGS, use the ASR instruction instead.

- Since characters in a SoftPLC STRING element are 2 bytes wide and serial communications often requires 8 bit characters, now an **encoding** parameter is used to specify how to convert from the 16 bit STRING characters to the serialized data stream. This is specifiable per port and the chosen encoding methodology applies to both sending and receiving characters, but receiving bytes is the reverse of sending bytes as far as the encoding methodology is concerned.

# Chapter 2. Warranty

Because of the variety of uses of the information described in this manual, the users of, and those responsible for applying this information must satisfy themselves as to the acceptability of each application and use of the information. In no event will SoftPLC Corporation be responsible or liable for its use, nor for any infringements of patents or other rights of third parties which may result from its use.

SOFTPLC CORPORATION MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR ANY PARTICULAR PURPOSE.

SoftPLC Corporation reserves the right to change product specifications at any time without notice. No part of this document may be reproduced by any means, nor translated, nor transmitted to any magnetic medium without the written consent of SoftPLC Corporation.

SoftPLC, and TOPDOC are registered trademarks of SoftPLC Corporation.

© Copyright 2006 SoftPLC Corporation ALL RIGHTS RESERVED

**First Printing**        November, 1994

**Last Printing**        January, 2006

SoftPLC Corporation 25603 Red Brangus Drive
Spicewood, Texas 78669
USA Telephone: 1-800-SoftPLC
Fax: 512/264-8399
URL: http://softplc.com
Email: support@softplc.com

# Chapter 3. Configuration

## 3.1. Module Editor

Use TOPDOC NexGen's PLC » Module Editor to load and configure the COMGENIUS.TLM for use with SoftPLC, as shown in the figure below. Clicking *Configure* opens the **configuration text editor**. Then press *Fetch* if remote editing, or *Load* if local editing to call up the respective copy of COMGENIUS.LST for editing.

*Sample COMGENIUS.LST file*

```
; The SoftPLC COMGENIUS.LST configuration file consist of 3 sections:
; [DRIVER], [PORTS], and [STRINGS].  Lines beginning with a semi-colon
; are considered comments and are ignored.  Anything after a semi-colon
; is a comment, unless it occurs in the [STRINGS] section.
; Each line in the [STRINGS] section is a format string, and must be
; bracketed in double quotes.  The first format string is considered index 1,
; the second format string is considered index 2, etc.

[DRIVER]
; Global, driver-wide settings

DEBUG=0

[PORTS]
; [PORTS] should include one line for each serial port you want active.
; The LOGICAL PORT value is what you give to the COMPRINT
; instruction's "Port" parameter.

; LOGICAL PORT
; |      PHYSICAL SERIAL PORT 0-31
; |      |       Encoding, usually "ISO-8859-1"
; |      |       |                       Baudrate: 9600 or 19200 or 38400
; |      |       |                       |       Databits: 8
; |      |       |                       |       | Parity: N or E or O
; |      |       |                       |       | | Stopbits: 1 or 2
; |      |       |                       |       | | | Timeout milliseconds
; |      |       |                       |       | | | |
; |      |       |                       |       | | | |
  0,    0,      ISO-8859-1,     38400,  8, N, 1, 500


[STRINGS]
1    "Rack no: %3o has %2d modules.\r\n"    example 1
2    "\x02\x03%.2C%.2C%#R"                   example 2 & 5, Modbus slave 2, function 3
3    "AK HM %.14s %.3d"                      example 3
4    "\x02\x03\xfa%.250C%#R"                 example 4 & 5
5    "\x02\x83%c%#R"                 example 5, Modbus exception response
6    "A STRPRINT test: number:%2d float:%g ST:%s"
```

```
7    "A_STRing -3.14 -4555"
8    "%s %f %d"
```

# 3.2. Configuration File

The following subsections describe the format of the configuration file COMGENIUS.LST

## 3.2.1. [DRIVER]

The [DRIVER] section holds driver wide configuration variables. Currently, only DEBUG is supported. Set it to zero to turn off debugging. Setting it to 1 or 2 will turn on debugging to varying degrees and cause COMGENIUS to print diagnostic information to the console as it operates.

## 3.2.2. [PORTS]

Each row of text in the [PORTS] section which is not a comment gives a port definition. A separate definition is needed for each port that will be used with COMGENIUS. There are 8 fields per row, described below:

| Field | Meaning |
| --- | --- |
| LOGICAL PORT | Each port definition should use a unique number in the range 0-31, normally starting at zero. The LOGICAL PORT value is what will be needed when a COMPRINT or COMSCAN instruction is entered and you are asked for the port parameter. |
| PHYSICAL COMPORT | The is COM1 to COM31. Physical com ports on a SoftPLC runtime have a prefix of COM and are numbered starting at 1. |
| Encoding | Set this to ISO-8859-1 if you are working with 8 bit characters like ASCII or Latin-1. Otherwise if you are working with characters outside the range of 0-255, see here. |
| Baudrate | Normally 9600 or 19200 or 38400 |
| Databits | May be 5, 6, 7, or 8. 8 is standard. |
| Parity | N = none O = odd E = even |
| Timeout | The time to wait for a COMSCAN instruction to complete, in milliseconds. COMSCAN gives up after this time duration. |

## 3.2.3. [STRINGS]

The string table is a read-only array of strings. As read-only, the strings may not be modified at run time. Strings from the string table are used solely as format specifications. The string table is edited

with the same configuration text editor as is used for the other configuration file sections.

> The strings you add to the string table are numbered from 1. COMGENIUS supplies string number 0 itself, and it is the empty string: "" . Remember, your first string is numbered 1.

A string table may contain up to 32,000 strings, or up to the limits of available RAM memory in the runtime, whichever comes first. A single string may not exceed 255 characters in length.

When loading the RAM resident string table from the COMGENIUS.LST file, each line of text within the file is scanned for a string delineated with a starting and ending double quote character. Text which is not enclosed in double quotes is ignored. Text outside double quotes may be used to document or comment the associated string. If no double quotes are found on a given line of text, then that string will be equivalent to the null string at runtime. The null string is the same thing as "".

**Escape Sequences**

If a string needs to make use of the " character itself, it should be preceded with the escape delineation character \. The \ character is also used to specify non-ASCII (extended) bytes and control codes. If the \ character itself is needed within a string, it must be preceded with another instance of \. The full list of supported escape sequences is given in the following table. All escape sequences are translated to 1 character at the time of string table load. In other words, these sequences look like multiple characters, but represent only one.

In addition, an arbitrary character value can be specified by \ooo, where ooo is one to three octal digits (0...7) or by \xHH, where HH is one or two hexadecimal digits (0...9, a...f, A...F), or by \uHHHH, where HH is one to four hexadecimal digits.

| Escape Sequence | Meaning |
| --- | --- |
| \a | alert (bell character), same as \x07 |
| \b | backspace, same as \x08 |
| \f | formfeed, same as \x0C |
| \n | line feed, same as \x0A |
| \r | carriage return, same as \x0D |
| \t | horizontal tab, same as \x09 |
| \v | vertical tab, same as \x0B |
| \\ | backslash |
| \" | double quote |
| \ooo | **octal escape sequence** which is an 8 bit character value given in octal, where each 'o' is a different octal digit, 0-7. For example \02 |
| \xHH | **hex escape sequence** which is an 8 bit character value given in hexadecimal |

| Escape Sequence | Meaning |
|---|---|
| \uHHHH | **unicode escape sequence** which is a 16 bit character value given in hexadecimal |

Strings in the string table are stored as a sequence of 16 bit characters. The only way to set the most significant byte of any character is to use the \uHHHH form for each such character whose upper byte is not zero. For ASCII characters, the upper byte will be zero. The first and last double quotes encountered on each line define the string that will be retained in the RAM resident string table, enclosing double quotes will be stripped off.

Here is a sample string table which is shown with its [STRINGS] section identifier.

```
[STRINGS]
1 "Enter your \"name:\""          prompt the user for his name
2 "Enter your access code:\a"     prompt user for access code and beep (\a).
3 Tell user thank you.            "Thank you for being there."
4 "\x1B[%d;%dH"                   ANSI terminal cursor position code,
5 which is ESC [#;#H
6 "\x10\x06"                      Data Highway DLE ACK control codes.
7 "NK %S\0"                       Tool Coordinator Not Acknowledge
8 "HM %3d %d %S\0"                Tool Positioner Home command.
"\b\b\b"                   9      3 backspaces to erase last input.
"First line\r\nsecond line"   10  Output two lines using \r\n.
"Here is a unicode character: \u124F"
```

The [STRINGS] section file will generate 11 strings, numbered 1 to 11 in the RAM resident string table. String 5 will be the null string, since line 5 has no double quoted text (line 5 was used as a continuation of a comment relating to string 4).

The above example shows how COMGENIUS looks only for the first and last set of double quotes to define the bounds of each string. (See lines 3, 9 and 10).

# Chapter 4. Usage

This section contains detailed setup and reference material. Each of the **TLIs** are described in a good bit of detail. If this is your first reading of this section, you may want to merely scan it. Then study the Examples section in detail, and while doing so, come back to this section treating it as reference material.

## 4.1. Installation

The TLM is named comgenius.tlm.so and is found as part of the standard SoftPLC 4.x installation in the /SoftPLC/tlm directory. To use it you merely have to enable it in NexGen's PLC | MODULES editor by enabling the *Use* checkbox in the same row as the COMGENIUS TLM.

Then you must edit the text file COMGENIUS.LST which is the TLM's configuration file. There is a text editor for this COMGENIUS.LST file within NexGen. It is easy to edit the configuration file from the PLC | MODULES editor. Simply click on the ***Configure*** button after selecting and enabling *Use* in the same row as the COMGENIUS TLM.

## 4.2. Editor Usage

```
PLC  COMGENIE's  COMGENIUS.LST                    _ □ X

Load    Save                                  Fetch    Send

; The SoftPLC COMGENIUS.LST configuration file consist of 3 sections:
; [DRIVER], [PORTS], and [STRINGS].  Lines beginning with a semi-colon
; are considered comments and are ignored.  Anything after a semi-colon
; is a comment, unless it occurs in the [STRINGS] section.
; Each line in the [STRINGS] section is a format string, and must be
; bracketed in double quotes.  The first format string is considered index 1,
; the second format string is considered index 2, etc.

[DRIVER]
; Global, driver-wide settings

DEBUG=0


[PORTS]
; [PORTS] should include one line for each serial port you want active.
; The LOGICAL PORT value is what you give to the COMPRINT
; instruction's "Port" parameter.

; LOGICAL PORT
; |      PHYSICAL COMPORT COM1-COM32
; |       |        Encoding, usually "ISO-8859-1"
; |       |              |             Baudrate: 9600 or 19200 or 38400
; |       |              |                |       Databits: 8
; |       |              |                |       | Parity: N or E or O
; |       |              |                |       | | Stopbits: 1 or 2
; |       |              |                |       | | | Timeout milliseconds
; |       |              |                |       | | | |
; |       |              |                |       | | | |
; |       |              |                |       | | | |
  0,     COM1,  ISO-8859-1,          38400,  8, N, 1, 1200


[STRINGS]
"string C5:0.ACC=%04X F8:0=%g\n"
"HELLO Mr. %s, This is your favorite PLC calling.\n"
"%.7s"
```

*Fetch, Send, Load, and Save* all have the same meaning as they do in the NexGen Module editor. You can see the helpfile for that editor by going to that editor and clicking on *Help*.

Use *Send* to transfer the configuration down to the SoftPLC. The next step is to cycle power on the SoftPLC for the changes to take place.
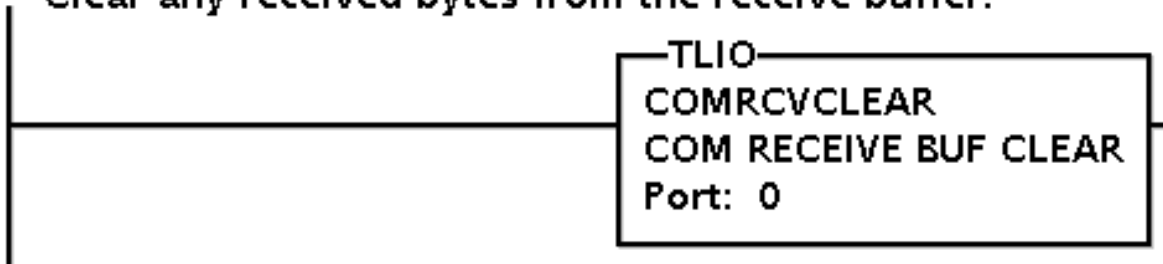
# 4.3. Ladder Instructions

This TLM implements the following TLIs (ladder instructions).

### 4.3.1. COMRCVCLEAR

This TLI throws away any bytes which are in the receiver buffer for the specified Port that have yet to be read in by COMSCAN. After this instruction is energized, no further input is possible with COMSCAN until more bytes actually arrive through the Port on the wire.
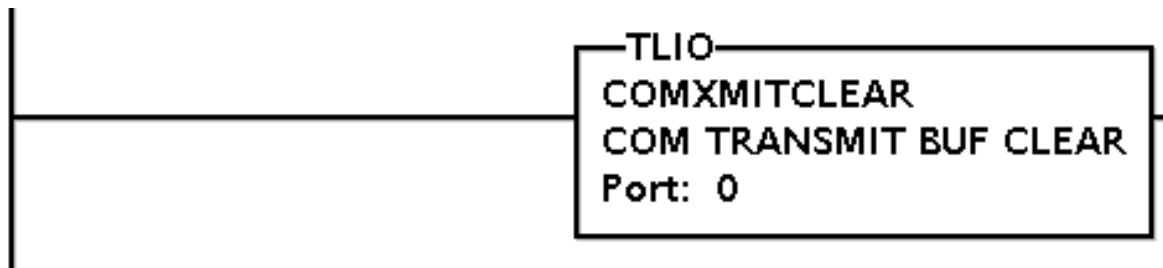
```
                  Clear any received bytes from the receive buffer.
                                          ┌─TLIO─────────────────┐
                                          │ COMRCVCLEAR          │
                                          │ COM RECEIVE BUF CLEAR│
                                          │ Port: 0              │
                                          └──────────────────────┘
```

| Parameter | Meaning |
| --- | --- |
| Port: | The LOGICAL PORT configured in COMGENIUS.LST |

## 4.3.2. COMXMITCLEAR

This TLI throws away any bytes in the transmitter buffer for the specified Port. After this instruction is energized, no further output will take place until a COMPRINT instruction is energized for this Port.
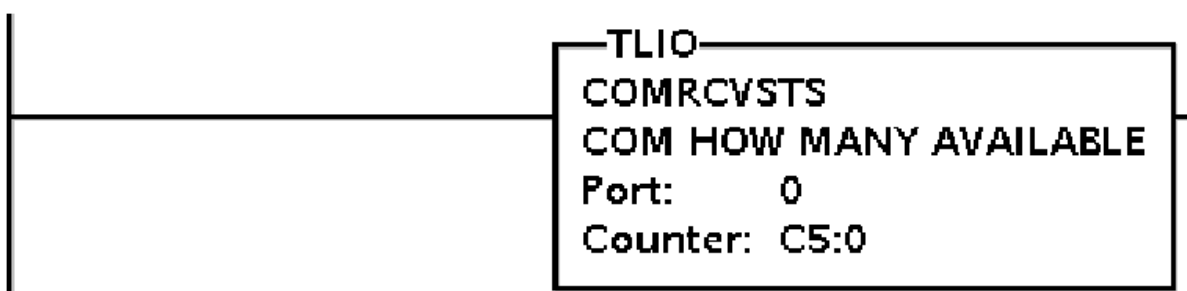
```
                                          ┌─TLIO─────────────────┐
                                          │ COMXMITCLEAR         │
                                          │ COM TRANSMIT BUF CLEAR│
                                          │ Port: 0              │
                                          └──────────────────────┘
```

| Parameter | Meaning |
| --- | --- |
| Port: | The LOGICAL PORT configured in COMGENIUS.LST |

## 4.3.3. COMRCVSTS

This TLI determines the number of characters available within the receive buffer for a particular Port.

The PRE field of the Counter is set to the size of the receive buffer. The ACC word is set to the count of received characters in the buffer. The overflow bit (OV) of the counter is set if there has been a receive buffer overflow. This internal flag is automatically reset during the next COMSCAN which removes enough characters to correct the overflow condition.

```
                                          ┌─TLIO─────────────────┐
                                          │ COMRCVSTS            │
                                          │ COM HOW MANY AVAILABLE│
                                          │ Port:     0          │
                                          │ Counter: C5:0        │
                                          └──────────────────────┘
```

| Parameter | Meaning |
| --- | --- |
| Port: | The LOGICAL PORT configured in COMGENIUS.LST |
| Counter: | Where to put the received characters count as described above. |

### 4.3.4. COMPRINT

COMPRINT outputs a packet of characters which is dynamically assembled using powerful string formatting. COMPRINT is a general purpose output formatting function, similar in behavior to the C programming language's printf() function, but extended with automatic checksum generation capabilities. COMPRINT can print binary byte packets, not merely ASCII strings. For example, to print out a byte whose value is zero, simply put it into the format string with an <u>octal escape sequence</u> like this: \0.
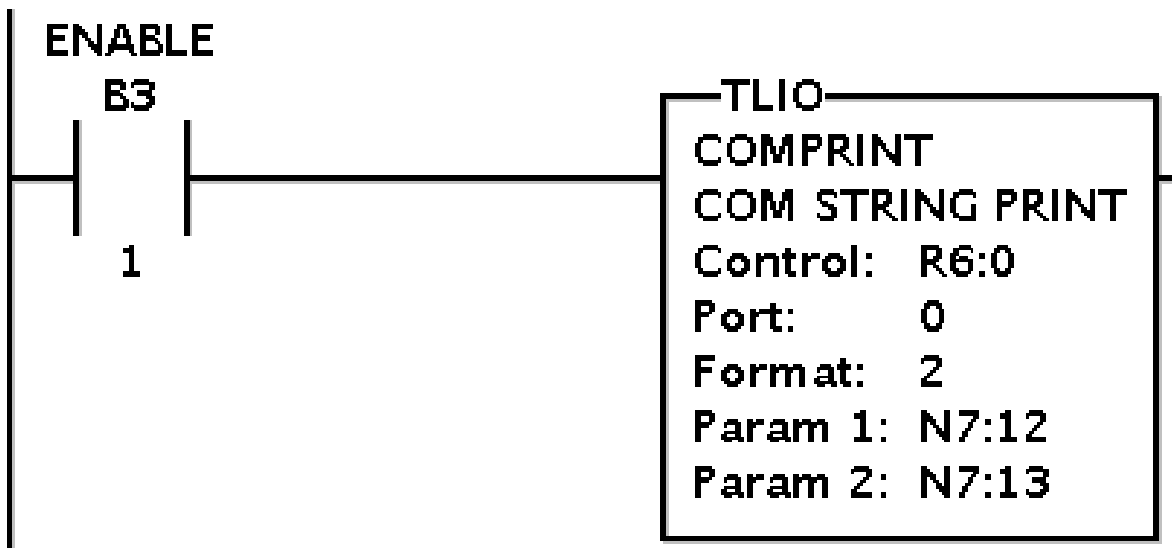


| Parameter | Meaning |
| --- | --- |
| Control: | A datatable CONTROL element of type R which is used to track the progress of the print operation. |
| Port: | The LOGICAL PORT configured in COMGENIUS.LST |
| Format: | This is an integer index into the <u>format string table</u>. **The first string is numbered 1, not zero.** The chosen format string is the master specification for the conversion, as explained in detail below. |
| Param 1 to 6: | These are 6 optional parameters. Each one may supply either a STRING element, FLOAT word(s), or INTEGER word(s) for inclusion in the conversion as explained in detail below. |

COMPRINT converts and writes output to the Port under control of the string table string given by

Format. The format string consists of a (potentially repeating) sequence of two types of objects: (a) ordinary characters, which are copied to the output Port verbatim, and (b) **conversion specifications**.

A conversion specification causes conversion and output of the next successive Param given to COMPRINT. Each conversion specification begins with the character % and ends with a conversion character. To output a %, use a %%.

A **conversion specification** consists of:

```
%[flag][width][.precision][l or L]conversion_char
```

where a % always signifies the beginning of a conversion specification.

> The [] brackets are illustrative only and are not part of the conversion specification. Fields wrapped in angle brackets are optional.

As shown above, between the % and the conversion_char there may be, in order, the following fields:

| Field Name | Description |
|---|---|
| flag | one or more of the following flags, which modify the specification:<br><br>- : which specifies left adjustment of the converted argument in its field.<br><br>+ : which specifies that the number will always be output with a sign.<br><br>**space** : if the first character is not a sign, a space will be prefixed.<br><br>**0** : for numeric conversions, specifies fill in the field with leading zeros up to the width.<br><br># : specifies an "alternate form", and its interpretation varies by conversion_char. For o, the first digit will be zero. For x or X, 0x or 0X will be prefixed to a non zero result. For e, E, f, g, and G, the output will always have a decimal point; for g and G, trailing zeros will not be removed. For S, a two's complement form (negated form) of checksum is generated instead of a simple positive summation. For R, the crc16 is computed by pre-loading the total with 0xFFFF as is required by Modbus RTU protocol. |

| Field Name | Description |
| --- | --- |
| width | is a sequence of decimal digits. Or it may be the character *, in which case the actual value is taken from the next integer Param and that Param is consumed. The interpretation of width varies based on conversion character:<br><br>S or R: See the S and R conversion characters below for details.<br><br>All others: minimum width of the field. The converted Param will be printed in a field at least this wide, and wider if necessary. If the converted Param has fewer characters than the field width it will be padded on the left (or right, if left adjustment has been requested) to make up the field width. The padding character is normally space, but is 0 if the zero padding *flag* is present. |
| ' . ' (period) | is used to separate the width from the precision, as seen in the **conversion specification** template above. |

| Field Name | Description |
|---|---|
| precision | is a sequence of decimal digits. Or it may be the character *, in which case the actual value is taken from the next integer Param and that Param is consumed. If no digits appear after the '.' (period), then the precision is taken as 0. If the '.' does not appear, no precision may be specified and it defaults to 1. If the precision is the * character, then the value is taken from the next Param which must be supplied as type integer, and this next Param is thusly consumed. The interpretation of precision varies based on conversion character:<br><br>e, E, or f: the number of digits to be printed after the decimal point.<br><br>g or G: the number of significant digits.<br><br>c or C: the number of consecutive characters to output starting with the supplied Param.<br><br>o, i, b, u, or d: minimum number of digits to be printed (leading 0s will be added to make up the necessary width).<br><br>s: the maximum number of characters to be printed from a STRING. In the event the STRING is longer than this number, only the first precision number of characters will be output. S or R: see the conversion characters S and R below for details. |

| Field Name | Description |
|---|---|
| l or L | means long length: when used with the o, u, x, X, i, d, or b conversion characters, l or L indicates that the corresponding Param is to be output as a 32 bit long integer in displayable form. For example, if N7:0 is passed as the Param for this conversion specification within the format string: **"%Ld"** then two integer words are used and interpreted as a 32 bit long integer N7:0 and N7:1, with the least significant 16 bits of the long word being N7:0. For all other conversion characters l or L is ignored. |

The last part of of the conversion specification is the **conversion_char**. It tells what to do with the corresponding Param in the TLI and it also locks down the **Required Type** for the Param:

| conversion_char | Required Param Type | Meaning |
|---|---|---|
| b | integer | output in displayable unsigned binary. |
| d | integer | output in displayable signed decimal. |
| i | integer | output in displayable signed decimal. |
| o | integer | output in displayable unsigned octal. |
| u | integer | output in displayable unsigned decimal. |
| x | integer | output in displayable unsigned hexadecimal, lowercase hex letters will be used. |
| X | integer | output in displayable unsigned hexadecimal, uppercase hex letters will be used. |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| s | STRING | The Param must be a STRING element. Each character in the STRING element is injected into the output by running it through the configured character encoding scheme. Use *width* to provide a minimum output field *width* if the input Param's STRING element contains a shorter number of characters, and padding will be output. Use *precision* to provide a maximum output field *width*. For example, a conversion specification of "%12.12s" will print exactly 12 characters. If the STRING is less than 12 it will be padded because the *width* is 12. If the STRING is less than 12 it will be truncated because the *precision* is 12. |

| conversion_char | Required Param Type | Meaning |
|---|---|---|
| c | integer | A single byte is taken from the least significant 8 bits of the integer Param and promoted to character by setting the upper 8 bits of the promoted unicode character to zero. Then the character is output with no conversion other than travelling through the normal configured output character encoding scheme, which if is ISO-8859-1 simply throws away the upper 8 bits of the character, outputting only the least significant byte of the character. The end result is that the original input byte is output unchanged.<br><br>If the *precision* is specified and is greater than 1, then the process above is repeated *precision* times, but each time with a different input byte. After the first byte, the output continues with the most significant 8 bits of the Param, then the least significant 8 bits of the word following the Param, then the most significant 8 bits of the word following Param, and so on, until the *precision* requirements are met. For example, if you wanted to send 6 bytes of binary information stored at N7:10 through N7:12 (2 bytes per word, least significant byte first, 3 words total), you would use the format specification "%.6c" and a Param of N7:10.<br><br>This is the little endian output. See 'C' (upper case) for big endian output. |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| C | integer | A single byte is taken from the most significant 8 bits of the integer Param and promoted to character by setting the upper 8 bits of the promoted unicode character to zero. Then the character is output with no conversion other than travelling through the normal configured output character encoding scheme, which if is ISO-8859-1 simply throws away the upper 8 bits of the character, outputting only the least significant byte of the character. The end result is that the original input byte is output unchanged.<br><br>If the *precision* is specified and is greater than 1, then the process above is repeated *precision* times, but each time with a different input byte. After the first byte, the output continues with the least significant 8 bits of the Param, then the most significant 8 bits of the word following the Param, then the least significant 8 bits of the word following Param, and so on, until the *precision* requirements are met. For example, if you wanted to send 6 bytes of binary information stored at N7:10 through N7:12 (2 bytes per word, most significant byte first, 3 words total), you would use the format specification "%.6C" and a Param of N7:10.<br><br>This is the big endian output. See 'c' (lower case) for little endian output. |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| f | float | decimal string of the form [-]dd.dddd. The number of digits after the decimal point is given by the *precision*, which defaults to 6. If the *precision* is 0, no fractional digits or decimal points appear. |
| e or E | float | string using scientific notation in the form [-]d.dddddde+-dd. There is one digit before the decimal point and *precision* digits after. The *precision* defaults to 6. If the *precision* is 0, the decimal point is not written. E is used for the exponent instead of e if the E conversion character was specified. A minimum of two digits will appear in the exponent. |
| g or G | float | string using either f or e (or E if G was specified) format, depending on the value of the Param. e will be used if the exponent is less than -3 or greater than the *precision*. The *precision* gives the number of significant digits; it defaults to 6. The decimal point appears if followed by a digit; trailing 0s are truncated. |
| % | no Param consumed | The % character is printed. |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| S | no Param consumed | The output is a binary checksum on all the characters output from this COMPRINT starting from the character at offset *width* and continuing up to and including the character *precision* characters before the place in the output stream where the first character of the checksum will go. *width* defaults to 0 and *precision* defaults to 1, which means sum all the characters preceding the checksum. If *precision* were 2, then the character before the checksum would be excluded, etc. A sixteen bit checksum is used (two bytes) and output little endian, unless the "h" Length modifier is present indicating a one byte checksum. The output from this conversion is not likely to be displayable. |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| S | no Param consumed | |

| conversion_char | Required Param Type | Meaning |
|---|---|---|
| R | no Param consumed | The output is a binary Cyclic Redundancy Check (CRC-16) on all the characters output from this COMPRINT starting from the character at offset *width* and continuing up to and including the character *precision* characters before the place in the output stream where the first character of the checksum will go. *width* defaults to 0 and *precision* defaults to 1, and 1 means include the character just before the checksum. If *precision* were 2, then the character before the checksum would be excluded, etc. A sixteen bit checksum is used, and is output little endian. The output from this conversion is not likely to be displayable.<br><br>This conversion char is useful for implementing the Modbus RTU protocol using COMGIENIUS. |

Any other conversion chars will cause an error.

**STATUS BITS/TROUBLESHOOTING COMPRINT**

The COMPRINT TLI uses a CONTROL parameter to execute asynchronously. The TLI starts its operation on a low to high rung transition. Unless and until the output buffer can hold the entire COMPRINT output, no partial output is performed. When the output is entered into the buffer, the DN bit comes on within the Control and the LEN is set to the number of bytes actually output. (Note that this may happen before the characters are actually transmitted out the physical Port.) The characters are in a ring buffer initially and the first one is or soon will be on its way out the physical Port. If the rung conditions go false before the TLI can get access to the sufficiently depleted Port buffer such that the entire output will fit, no output is performed.

There is no timeout associated with COMPRINT. At most 6 conversion Params and therefore at most 6 conversion_specifications (of the type requiring a Param) may be specified in this TLI. COMPRINT can fail to print only if:

1. you never energize the rung

2. you have a bad format string INDEX

3. you have not allowed the rung to go false from the last time you fired it and the DN bit came on

4. you have a bad port number

5. you have an illegal format string (for example, "%k")

> If any of the above error conditions are true, then the ER bit is set AND the DN bit is set. COMPRINT can be delayed if there is insufficient room in the output buffer. This is indicated by the DN bit not coming on.

If none of the conditions 1) through 5) happen, then you expect the DN bit to come on and the ER bit will be reset by the instruction. You do not ever have to reset the ER bit on COMPRINT because successful completion clears the ER bit. Additionally, a false rung condition clears both DN and ER [condition 1) above].

### 4.3.5. COMSCAN

COMSCAN reads bytes from the receive buffer for a port. Bytes read are first converted according to the *encoding scheme* to a batch of characters, and then the batch of characters are converted according to a format string and the values created are stored in the TLI's integer, float, and/or STRING element datatable Params. COMSCAN is similar to the C language "scanf" function, is a general purpose serial input function, and has very powerful and wide ranging **pattern matching** and conversion capabilities.
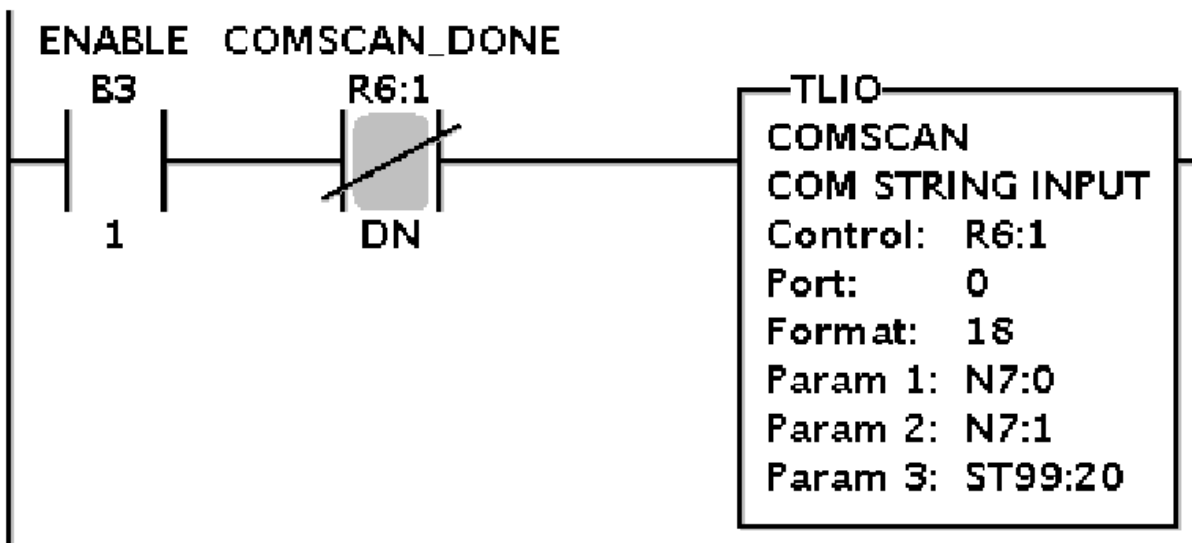


| Parameter | Meaning |
| --- | --- |
| Control: | A datatable CONTROL element of type R which is used to track the progress of the scan operation. |
| Port: | The LOGICAL PORT configured in COMGENIUS.LST |

| Parameter | Meaning |
| --- | --- |
| Format: | This is an integer index into the <u>format string table</u>. **The first string is numbered 1, not zero.** The chosen format string is the master specification for the conversion, as explained in detail below. |
| Param 1 to 6 | These are 6 optional parameters. Each one may receive either a STRING element, FLOAT word(s), or INTEGER word(s) as a result of parsing and converting fields in the batch of input characters, as explained in detail below. |

A major feature of this TLI, besides reading characters, is **pattern matching** on those characters according to the format string. The instruction either succeeds or fails based on the pattern matching, and this is indicated by the ER bit being on or off at the time the DN bit comes on.

> In order for the TLI to succeed, the entired format string must be matched. If the instruction does not succeed, then all characters tested during the pattern matching are put back into the receive buffer for use by the next invocation of the COMSCAN TLI.

The format string consists of a (potentially repeating) sequence of the following elements:

1. Spaces, tabs, carriage returns, line feeds, vertical tabs, and formfeeds which all cause input to be skipped up to the next character which is not whitespace. (Whitespace characters are blank, tab, line feed, carriage return, vertical tab, and formfeed.)

2. Other characters, except for % which are matched against the input.

3. A conversion specification:

```
%[* or #][width][.precision][modifier]conversion_char
```

where a % always signifies the beginning of a conversion specification.

> The [] brackets are illustrative only and are not part of the conversion specification. Fields wrapped in angle brackets are optional.

A conversion specification determines the conversion of the next input field. The result of the conversion is placed in the next Param. Params are thus consumed one after another in this fashion. If there are less Params than conversion specifications an error is returned. Excess Params are ignored. If the * is present, the conversion is performed, but no Param is consumed.

An input field is normally a string of non-white space characters; it extends either to the next white space character or until the field width, if specified, is exhausted. This implies that COMSCAN may read across line boundaries to find its input, since newlines are white space.

The conversion_char indicates the interpretation of the input field. A % always signifies the

beginning of a conversion specification. As shown above, between the % and the *conversion_char* there may be, in order, the following fields:

| Field Name | Description |
| --- | --- |
| * | is the assignment suppression indicator. Conversion verification and matching will take place, but no Param is consumed. |
| # | is the "alternate form" indicator, and its interpretation varies by conversion_char. For S, a two's complement form (negated form) of checksum is generated instead of a simple positive summation. For R, the crc16 is computed by pre-loading the total with 0xFFFF as is required by Modbus RTU protocol. |
| width | is a sequence of decimal digits. It may NOT be the '*' character. The interpretation of width varies based on conversion character:<br><br>S or R: See the S and R conversion characters below for details.<br><br>All others: ignored. |
| . (period) | is used to separate the width from the *precision*. |

| Field Name | Description |
| --- | --- |
| precision | is a sequence of decimal digits. It may NOT be the '*' character. If the period '.' does not appear, no *precision* may be specified and it defaults to a sensible value based on the *conversion_char*. If present, the default is overridden and the interpretation of *precision* varies based on *conversion_char*: |
| | S or R: See the S and R conversion characters below for details. |
| | c or C: Specifies the number of bytes to be copied from the raw input stream. |
| | s: Specifies the minimum number of characters to be copied into the STRING element. Matching and copying stop after *precision* characters, or if whitespace is encountered first. |
| | o, u, x, X, i, d or b: Specifies the minimum number of numerical digits to expect excluding the prefix like +,- or 0x. Matching and conversion stop after *precision* characters, or if a non-digit is encountered first. |
| | All others: ignored |
| modifier | may be one of L, l, H, or h. |
| | L or l: indicates that the corresponding argument is to be converted to a 32 bit long integer when used with the o, u, x, X, i, d or b *conversion_char*. For all other *conversion_chars* it is ignored. |
| | H or h: used only for the S *conversion_char*, see below. |

The last part of the conversion specification is the *conversion_char*. It tells how to convert the next non-whitespace field in the input character stream:

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| b | integer | unsigned binary, like "1011011" |
| d | integer | match and convert a signed decimal field, like "-423" or "423" or "+432". Using *precision* is normal, example: "%.3d" |
| i | integer | if the field starts with a zero (0), it is interpreted as octal, if it starts with 0x or 0X it is interpreted as hexadecimal, otherwise it is interpreted as decimal. Like "077" or "0xFF" or "12" |
| o | integer | unsigned octal, like "077" |
| u | integer | unsigned decimal, like "423" |
| x or X | integer | unsigned hexadecimal, like "AF9B" |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| | | unsigned binary, like "1011011" |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| c | integer | is a character copying input. *precision* number of input characters are copied verbatim starting at the destination integer Param. **Only the least significant 8 bits of each character are used.** *precision* defaults to 1. The lower 8 bits of each input character are copied and packed into the destination integer block using **little endian** copying, with up to 2 characters per integer word.<br><br>The first character is put into the least significant 8 bits of the integer Param, the **next** input character (if the *precision* is greater than 1) is put into the most significant byte of the integer Param, then the next input character is copied into the least significant byte of the next integer word after the Param, etc., until *precision* characters have been copied to the block of words given by the Param.<br><br>For example: "%.30c" would match and copy the next 30 characters from the input stream, regardless of their values, and would fill 15 integer words starting at Param.<br><br>Only the lower 8 bits of each character are used. This conversion_char effectively "matches" any character or *precision* number of characters. The lower 8 bits of each character are packed into the destination Param verbatim using **little endian** copying. |

| conversion_char | Required Param Type | Meaning |
|---|---|---|
| C | integer | is a character copying input. *precision* number of input characters are copied verbatim starting at the destination integer Param. **Only the least significant 8 bits of each character are used.** *precision* defaults to 1. The lower 8 bits of each input character are copied and packed into the destination integer block using **big endian** copying, with up to 2 characters per integer word. <br><br> The first character is put into the most significant 8 bits of the integer Param, the next input character (if the *precision* is greater than 1) is put into the least significant byte of the integer Param, then the next input character is copied into the most significant byte of the **next** integer word after the Param, etc., until *precision* characters have been copied to the block of words given by the Param. <br><br> For example: "%.30C" would match and copy the next 30 characters from the input stream, regardless of their values, and would fill 15 integer words starting at Param. <br><br> Only the lower 8 bits of each character are used. This conversion_char effectively "matches" any character or *precision* number of characters. The lower 8 bits of each character are packed into the destination Param verbatim using **big endian** copying. |

| conversion_char | Required Param Type | Meaning |
|---|---|---|
| e, E, f, g or G | float | floating point number is put into the Param. |
| %<br>' ' (space)<br>\t (tab)<br>\v (vertical tab)<br>\r (carriage return)<br>\n (line feed)<br>\f (form feed) | no Param consumed | Any **one** of the 7 conversion_chars may be given in the format string. *precision* is ignored. The given conversion_char is merely matched in the input stream of characters and **thrown away**. No Param is consumed.<br><br>Examples:<br>"% " matches a space<br>"%\t" matches a tab<br>"%\r" matches a carriage return |

| conversion_char | Required Param Type | Meaning |
|---|---|---|
| s | STRING | The Param is a STRING element like ST99:48 that receives the next **non-whitespace** block of input characters. Leading whitespace is ignored. The input characters are stored in the STRING element until: <br>• a whitespace character is encountered, or <br>• the STRING element is filled entirely (82 characters), or <br>• *precision* characters have been put into the STRING element. (*precision* defaults to 82.) <br><br>If the receive buffer runs out of input characters before any of the above 3 conditions are matched, then the matching fails. <br><br>All 16 bits of each character are saved into their respective character position within the STRING element. <br><br>If specific quantities of whitespace must be included in the input scan, here are some additional conversion_chars to consider: <br><br>1. use %c or %C with *precision*, for example: "%*.2c", will match *any* 2 characters and suppress assignment to a Param (i.e. throw away the next two characters). |
| s | | 2. use "% " (percent space), which matches a single space character and throws it away. See % above. |

| conversion_char | Required Param Type | Meaning |
| --- | --- | --- |
| S | no Param consumed | The width defaults to 0 and the *precision* defaults to 1. The output is a binary checksum on all the characters received (NOT the same as converted) during this COMSCAN starting from the character at offset *width* up to and including the character *precision* characters before this field.<br><br>The calculated value is compared to the two characters in the input at this position (or 1 byte if the 'h' *modifier* is given). |
| R | no Param consumed | The width defaults to 0 and the *precision* defaults to 1. The output is a binary Cyclic Redundancy Check (CRC-16) on all the characters input during this COMSCAN starting from the character at offset *width* and continuing up to and including the character *precision* characters before this field. A sixteen bit CRC-16 is used.<br><br>The initial value of the CRC is preloaded with zero (0), unless the # flag is used in which case the initial value is 0xFFFF. The # flag is necessary when receiving Modbus RTU replies.<br><br>The CRC-16 computed in this way must match the 2 input characters at this position in little endian fashion. See the MODBUS example in the examples section. |
| n | integer | grabs the number of characters read up to this point. |

| conversion_char | Required Param Type | Meaning |
|---|---|---|
| All other characters are not conversion characters and will cause an error. | | |

The COMSCAN TLI uses a Control parameter to execute asynchronously. The TLI starts its operation on a low to high rung condition transition. When the TLI finishes the DN (done) bit in the CONTROL element is set. If there is an error at completion, then also the ER (error) bit is set, else not. After the start of operation on the low to high rung transition, the following behavior is expected for each scan:

| Condition | CONTROL and Status |
|---|---|
| Sufficient input characters have been read to match all the conversion specifications in the format string. | Successful Completion.<br><br>DN: set<br>ERR: not set<br>LEN: set to total number of characters read<br>POS: set to number of conversion specs matched |
| Sufficient input characters have been read to determine that the matching of conversion specifications cannot succeed. The invocation fails **and all input characters are put back into the receive** buffer for another <u>COMSCAN</u> or <u>COMRCVCLEAR</u>.<br><br>Read characters are put back into the receive buffer for next scan. | Failed Completion.<br><br>DN: set<br>ER: set<br>LEN: set to total number or characters read and indicates the offset into the input stream where the mismatch occurred<br>POS: set to number of conversion specs matched |
| Insufficient characters are available in the receive buffer to complete a match, and the **timeout has expired** as set in the configuration file for this port. The time is measured starting from the low to high rung transition.<br><br>Read characters are put back into the receive buffer for next scan. | Failed Completion.<br><br>DN: set<br>ER: set<br>EM: set<br>LEN: set to total number of characters read<br>POS: set to number of conversion specs matched |
| Insufficient characters are available in the receive buffer to determine a match, and the timeout has yet to expire.<br><br>Read characters are put back into the receive buffer for next scan. | Not Completed. As long as the instruction stays energized by the rung, the instruction waits for further character input or the timeout<br><br>DN: not set<br>ER: not set<br>LEN: set to total number of characters read<br>POS: set to number of conversion specs matched |

| Condition | CONTROL and Status |
|---|---|
| If the rung is de-energized before completion. | No Completion will occur.<br><br><br>EN: not set<br>DN: not set |

If the amount of input characters is insufficient to process the entire format string, and as long as the TLI remains energized by the rung, the instruction **waits for several scans** until enough data is available or until the Port specific TIMEOUT has expired.
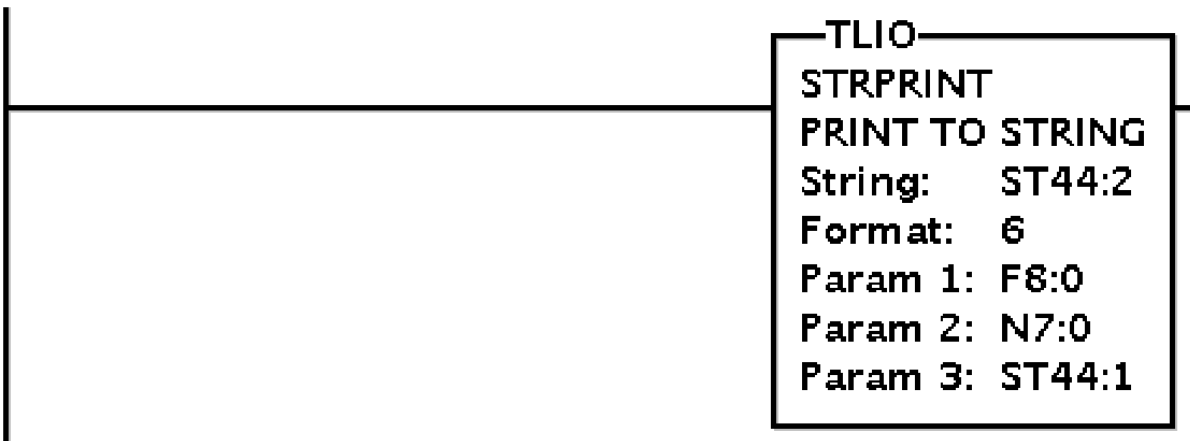
> For each scan for which the TLI is energized, any or all the datatable Params are subject to modification. This is true until the DN bit comes on, and whether or not the TLI ultimately succeeds or not. It is usually necessary to treat the Params as "staging areas" only that must be snap-shotted (copied) to a safe place at the moment the DN bit comes on and the ER bit is not on. Otherwise their values are not final nor reliable.

COMSCAN can fail to scan if any of the following are true:

1. you never energize the rung, or de-energize it before the timeout expires

2. you have a bad format string INDEX

3. you have not allowed the rung to go false from the last time you fired it and the DN bit came on

4. you have a bad port number

5. you have an illegal format string (for example, "%k")

6. you have a mismatch in the number of parameters indicated in the format string relative to the number supplied to the instruction, or a type mismatch on any parameter. (Eg: "%u" and then supply a float parameter to the TLI instruction, where %u requires an integer.)

7. a mismatch on the inbound data occurred, meaning the serial line data did not agree with the format string

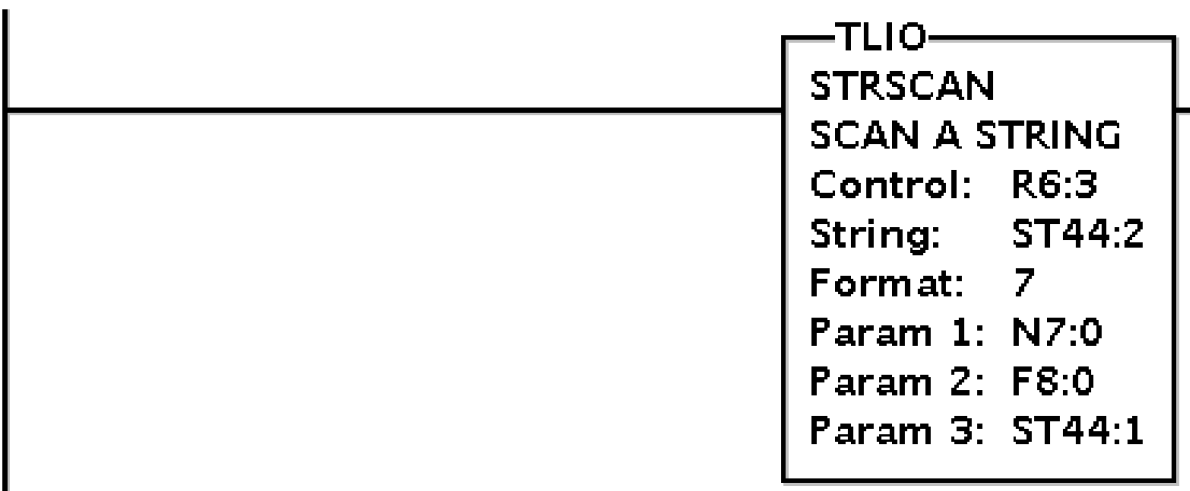8. a timeout occurred. This sets the EM, ER, and DN bits.

## 4.3.6. STRPRINT

STRPRINT outputs a packet of characters which is dynamically assembled using powerful string formatting. STRPRINT is almost exactly like COMPRINT, except that the output is to a STRING element and not to a serial port, and this instruction executes to completion immediately (is not asynchronous). Read about COMPRINT and then return here.

```
                              ┌─TLIO──────────┐
                              │ STRPRINT       │
                              │ PRINT TO STRING │
                              │ String:    ST44:2│
                              │ Format:    6     │
                              │ Param 1: F8:0    │
                              │ Param 2: N7:0    │
                              │ Param 3: ST44:1  │
                              └────────────────┘
```

| Parameter | Meaning |
|---|---|
| String: | Where to print to. Up to 82 characters may be copied into this STRING element. |
| Format: | This is an integer index into the <u>format string table</u>. **The first string is numbered 1, not zero.** The chosen format string is the master specification for the conversion, as explained in detail below. |
| Param 1 to 6 | These are 6 optional parameters. Each one may supply either a STRING element, FLOAT word(s), or INTEGER word(s) for inclusion in the conversion as explained in the COMPRINT detail. |

### 4.3.7. STRSCAN

STRSCAN reads characters from a STRING element. The characters are converted according to a format string and the values created are stored in the TLI's integer, float, and/or <u>STRING element</u> datatable Params. COMSCAN is similar to the C language "scanf" function, is a general purpose serial input function, and has very powerful and wide ranging pattern matching and conversion capabilities.

```
                              ┌─TLIO──────────┐
                              │ STRSCAN        │
                              │ SCAN A STRING  │
                              │ Control:  R6:3  │
                              │ String:   ST44:2│
                              │ Format:   7     │
                              │ Param 1: N7:0   │
                              │ Param 2: F8:0   │
                              │ Param 3: ST44:1 │
                              └────────────────┘
```

| Parameter | Meaning |
|---|---|
| Control: | A datatable CONTROL element of type R which is used to track the progress of the scan operation. |
| String: | The source of the input characters. Up to 82 characters may be read from this STRING element. |
| Format: | This is an integer index into the format string table. **The first string is numbered 1, not zero.** The chosen format string is the master specification for the conversion, as explained in detail below. |
| Param 1 to 6 | These are 6 optional parameters. Each one may receive either a STRING element, FLOAT word(s), or INTEGER word(s) as a result of parsing and converting fields in the input characters, as explained in detail at COMSCAN. |

Study the operation of the COMSCAN TLI to better understand how STRSCAN works. They are the same except for the source of the characters.

# Chapter 5. Debugging Tips

This section gives tips on debugging problems with this TLM.

## 5.1. Enabling Debug Prints

In the configuration file there is the DEBUG setting. It may be set to 0, 1, or 2 to indicate that you want no, some, or most debugging respectively. Remember a DEBUG value of "0" means no debugging. On version 4.x SoftPLC, all process output from the SoftPLC runtime engine is normally directed to the syslog, because SoftPLC runs as a daemon normally. The syslog can be configured in a number of different ways, but the default uses a small RAM resident FIFO and eventually will run out of space and wrap back around on itself. Rather than reconfiguring the syslog, there is an easier way.

Following is a procedure to get the debugging output into a text file.

1. Log into SoftPLC using either a) PUTTY from Windows or b) using ssh from Linux or c) at the command prompt of the SoftPLC system.

2. Run this command:
   # /etc/init.d/softplc.sh stop

3. Change into the /SoftPLC/run directory:
   # cd /SoftPLC/run

4. You can run SoftPLC from the command prompt now and redirect its output to an arbitrary file (named out.txt here). We put that file into the RAM disk which is anchored in the /tmp directory.
   # ./runsplc > /tmp/out.txt

5. Let this run for 5-60 seconds, then press control-C. Now you have the output captured in file /tmp/out.txt, each request-response transaction will be captured in that file.

6. You can look at the file using the program named "less".
   # less /tmp/out.txt
   You can look at this output with the Modbus Specification, and the manual for your Modbus master software in hand. Press ESC when done.

7. When done, remember to set debug back to "0", then you can start SoftPLC as a daemon either by a) power cycling the box or b) doing the following:
   # /etc/init.d/softplc.sh start

# Chapter 6. Examples

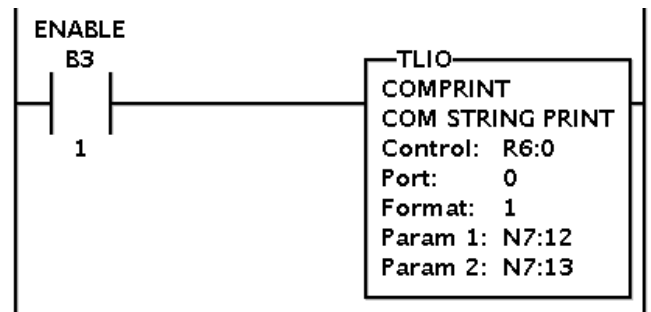## 6.1. Example 1. COMPRINT of Text with Integers

All examples below assume the following string table:

```
1 "Rack no: %3o has %2d modules.\r\n"   example 1
2 "\x02\x03%.2C%.2C%#R"                 example 2, Modbus slave 2, function 3
3 "AK HM %.14s %.3d"                    example 3
4 "\x02\x03\xfa%.250C%#R"               example 4, Modbus response
5 "\x02\x83%c%#R"                       example 5, Modbus exception
```

### 6.1.1. Example 1. COMPRINT of Text with Integers

| | |
|---|---|
| Format: 1<br>Two parameters: integer, integer |  |
| Sample output: "Rack no: 5 has 2 modules." | |

Look at our example string table's format string number 1. The trailing carriage return and line feed are not shown in the Sample output because they are not visible characters. There are two conversion specifications and therefore two Params are required, in this case both integers. The first conversion specification is %3o and means print out a field 3 characters wide, pad on the front end with spaces if needed and do it in octal. The second conversion specification is %2d and means print out a field 2 characters wide at least, pad on the front end with spaces if needed and do it in decimal.

## 6.2. Example 2. COMPRINT of Modbus RTU Query

In this example binary is printed rather than ASCII, so our format string has hex escapes. The desired Modbus RTU query is *Read Holding Registers*, Modbus function number 3 and intended for slave 2:

```
+-------+-------+-------+-------+-------+-------+-------+-------+
|  02   |  03   | Ref Hi| Ref Lo| Num Hi| Num Lo| CRC Lo| CRC Hi|
+-------+-------+-------+-------+-------+-------+-------+-------+
  slave   func  |<----2 bytes-->|<----2 bytes-->|<----2 bytes-->|
```

The 2 *Ref* bytes are the datatable address starting at 0 (which corresponds to register 40001), in most significant byte first format **(big endian)**. The 2 *Num* bytes are the count of registers to read, in big endian format, with a maximum of 125. The 2 *CRC* bytes are the CRC-16 calculated by preloading with 0xFFFF, not 0, and are output in little endian format.

| | |
|---|---|
| Format: 2<br>Two parameters: integer, integer | ENABLE<br>B3<br><br>1<br><br>TLIO<br>COMPRINT<br>COM STRING PRINT<br>Control: R6:0<br>Port: 0<br>Format: 2<br>Param 1: N7:12<br>Param 2: N7:13 |
| Sample output: see above where the 8 byte query is shown | |

Look at our example string table's format string number 2. The first byte is 0x02 and is the slave id we arbitrarily chose for this example. It could be any slave id and it could also have been created as a result of a %c (single byte) conversion specification. The next byte is hard coded as 0x03, the Modbus function. The next two bytes are a big endian 16 bit binary integer containing the memory address, so %.2C is the conversion specification. The next two bytes are a big endian 16 bit binary number containing the desired register count, so %.2C is the conversion specification. The datatable value for this, Param 2, cannot exceed 125 according to the Modbus specification. The last two bytes are the little endian CRC-16 using 0xFFFF as the CRC preload. To get the 0xFFFF preload, the alternate form specifier # is used in the conversion specification %#R. There is yet a simpler format string possible for this example, and that would be to hard code the Ref and Num words using 4 hex escapes, something that makes more sense if they do not need to be modifiable at runtime. In such a case no Params would be required, since the %#R conversion specification uses no Params itself.
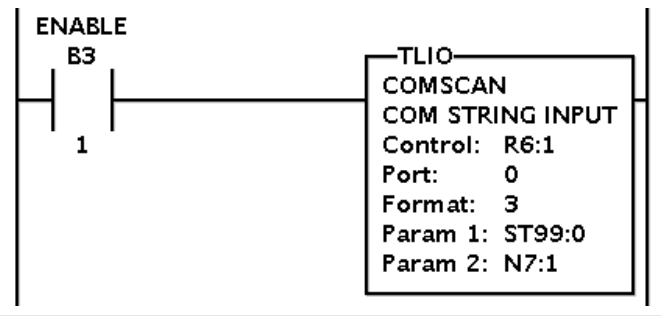
# 6.3. Example 3. COMSCAN of Text with Integers

In this example a string is expected on the Port in ASCII. The expected string looks like:

```
AK HM <variableWidthNonWhitespace> NNNCRLF
```

The AK HM are fixed. The <variableWidthNonWhitespace> is a field that will **not** have whitespace in it. This field's width will vary from 1 to 14 characters. The NNN is a decimal number like "123" and it is at most 3 characters wide. The CRLF are two characters: \r and \n respectively. They are whitespace according to our earlier definition.

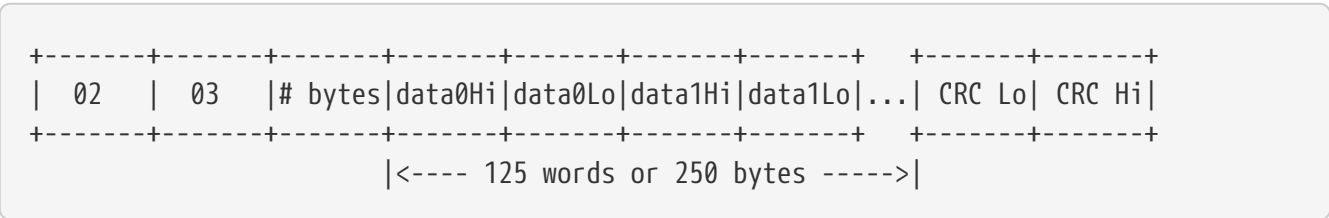| Format: 3<br>Two parameters: STRING element, integer | ENABLE<br>B3<br>┤├<br>1<br><br>─TLIO─<br>COMSCAN<br>COM STRING INPUT<br>Control:  R6:1<br>Port:      0<br>Format:   3<br>Param 1: ST99:0<br>Param 2: N7:1 |
| --- | --- |

Look at our example <u>string table's</u> format string number 3. The AK HM are fixed and must be matched explicitly. The first conversion specification is %.14s, meaning match and copy a non-whitespace string segment of up to 14 characters into the corresponding Param 1, which **must be** a STRING element. The copying will end at the space character preceding the NNN. The next conversion specification is %.3d, which means expect a decimal integer field (possibly with a leading '-' character) up to 3 digits wide. The end of the NNN field will be marked by whitespace (in this case the CRLF), or by any character after the 3rd decimal digit character because of the *precision* of 3.

**You almost always want to use *precision* in your COMSCAN conversion specifications, otherwise matching can erroneously succeed before all the bytes are received on the serial port.**
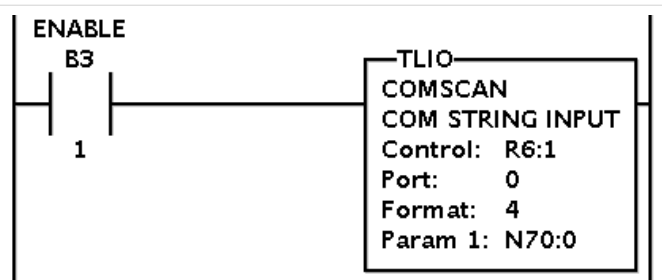
# 6.4. Example 4. COMSCAN of Modbus RTU Response

In <u>example 2</u>, COMPRINT was used to send a Modbus query. In this current example, the desired Modbus RTU response is for the *Read Holding Registers* query, Modbus function number 3 and coming back from slave 2:

```
+-------+-------+-------+-------+-------+-------+-------+   +-------+-------+
|  02   |  03   |# bytes|data0Hi|data0Lo|data1Hi|data1Lo|...| CRC Lo| CRC Hi|
+-------+-------+-------+-------+-------+-------+-------+   +-------+-------+
                        |<---- 125 words or 250 bytes ----->|
```

For discussion, we'll assume the query asked for 125 registers. Then the # bytes field will be 250 which is 0xFA in hex. COMSCAN is both a matcher and converter. If slave id, function, # bytes, or CRC do not match, then the ER bit is set along with the DN bit. However, the Params can still be modified even on COMSCAN mismatch and failure. **Therefore it is critical to buffer the Params and copy them to a safe useable place when the DN bit is on without the ER bit.**

| Format: 4<br>One parameter: integer | ENABLE<br>B3<br>┤├<br>1<br><br>─TLIO─<br>COMSCAN<br>COM STRING INPUT<br>Control:  R6:1<br>Port:      0<br>Format:   4<br>Param 1: N70:0 |
| --- | --- |

Look at our example string table's format string number 4. The first byte is 0x02 and is the slave id we arbitrarily chose for this example. It could be any slave id. The next byte is hard coded as 0x03, the Modbus function. The next byte is 0xFD which is 2 x 125, the byte count. Then comes the conversion specification %. 250C, which means convert 250 bytes in big endian fashion and store the results starting at Param1, with 2 bytes per integer. The last conversion specification is the %#R which is for the CRC-16, preloaded with 0xFFFF and compared little endian to the received bytes at this position. This conversion specification is for matching only, no conversion is stored in any Param, so no Param is required for it.

# 6.5. Example 5. Simple Modbus RTU Master

This example is a complete working Modbus RTU master implemented with a COMPRINT and two COMSCANs. The COMPRINT is basically described by example 2. On the serial cable, in response to the COMPRINT one of three results are expected:

1. A valid Modbus **response** packet

2. A valid Modbus **exception** packet

3. Garbage or no response at all

Matching a Modbus response was discussed in example 4, and it requires its own COMSCAN instruction with a taylor made format string. Matching a Modbus exception packet requires another COMSCAN instruction with a different taylor made format string. Because COMSCAN puts back all characters when any part of the matching process fails, this leaves them all available for the next COMSCAN to try and match against. If the second one also fails, any garbage characters can be cleared out of the receive buffer with the COMRCVCLEAR TLI.

The example is available both as a PDF printout and as a binary SOFTPLC.APP file. You can download either from our website. To use the SOFTPLC.APP file, put it into a new directory like \SoftPLC\app\MODBMAST\ or similar so that TOPDOC NexGen can find it. Acrobat Reader is needed to view the PDF file, and can also be used to print it out to a printer.

This example uses format strings 2, 4, and 5. Format strings 2 and 4 were discussed in examples 2 and 4 respectively. Format string 5 is designed to match an exception response, per the Modbus specification and capture the single byte exception code into the least significant byte of an integer Param.

Please read the PDF file and the rung comments for the full understanding of the program logic.