

# **PACSystems™**

C - Programmer's Toolkit

User's Manual

# Contents

<b>Chapter 1: Introduction .....</b>	<b>1</b>
1.1 Revisions in this Manual .....	1
1.2 Documentation .....	2
<b>Chapter 2: Installation .....</b>	<b>3</b>
2.1 System Requirements .....	3
2.2 Installing the C Toolkit for PACSystems .....	3
2.2.1 To install the Toolkit .....	4
2.3 Running C Toolkit .....	5
2.4 C Toolkit File Structure .....	5
2.4.1 Directories .....	6
2.4.2 Files .....	6
2.5 Uninstalling C Toolkit .....	6
<b>Chapter 3: Writing a C Application .....</b>	<b>7</b>
3.1 Name Requirements .....	7
3.1.1 File Names .....	7
3.1.2 Reserved Names .....	8
3.2 C Applications in the PACSystems Environment .....	8
3.2.1 Developing a C Block .....	8
3.2.2 C Toolkit Variable Types .....	10
3.2.3 Compiling .....	12
3.2.4 Associating a Compiled C Block with the Application Program .....	16
3.2.5 Adding Blocks through the Machine Edition Programmer .....	16
3.2.6 Specifying Parameters .....	17
3.2.7 Scheduling C Blocks .....	17
3.2.8 Using a C Block in an LD or FBD Program .....	18
3.2.9 Using a C Block in an ST Program .....	18
3.3 PACSystems C Block Structure .....	19
3.3.1 Variable Declarations .....	20
3.3.2 Stack Overflow Checking .....	20
3.3.3 Parameter Pointer Validation .....	23
3.4 PLC Reference Memory Access .....	23
3.4.1 How to Format a PLC Reference Access Macro .....	25

3.4.2	Bit Macros .....	25
3.4.3	Byte Macros .....	27
3.4.4	Integer/Word Macros .....	28
3.4.5	Double Word/Floating Point Macros .....	28
3.4.6	Double Precision Floating Point Macros .....	29
3.4.7	Reference Memory Size Macros .....	29
3.4.8	Transition, Alarm, and Fault Macros.....	30
3.5	Standard Library Routines.....	32
3.5.1	PACSystems Functions .....	32
3.5.2	General PLC Functions.....	33
3.5.3	Bus Read/Write Functions .....	39
3.5.4	BUS Semaphore Functions .....	48
3.5.5	Service Request Functions .....	53
3.5.6	Fault Table Service Request Functions .....	79
3.5.7	Module Communications .....	88
3.5.8	Ladder Function Blocks.....	89
3.5.9	Miscellaneous General Functions.....	93
3.5.10	Reference Memory Functions .....	95
3.5.11	Utility Function.....	111
3.5.12	Errno Functions .....	112
3.5.13	PLC Variable Access .....	113
3.6	Application Considerations .....	131
3.6.1	Application File Names .....	131
3.6.2	Floating Point Arithmetic .....	131
3.6.3	Available Reference Data Ranges.....	131
3.6.4	Global Variable Initialization .....	133
3.6.5	Static Variables.....	134
3.6.6	Data Retentiveness .....	134
3.6.7	GefMain( ) Parameter Declaration Errors for Blocks.....	135
3.6.8	Uninitialized Pointers .....	139
3.6.9	PLC Local Registers (%P and %L) .....	140
3.6.10	Block Enable Output (ENO).....	141
3.6.11	Writes to %S Memory Using SB(x) .....	142
3.6.12	FST_EXE and FST_SCN Macros .....	142
3.6.13	LST_SCN Macro .....	142

3.6.14	Runtime Error Handling .....	143
3.6.15	C Application Impact on PLC Memory .....	143
3.6.16	Blocks as Timed or I/O Interrupt Blocks .....	144
3.6.17	Restricting Compilation to a Specific Target .....	145
<b>Chapter 4:</b>	<b>Debugging and Testing C Applications .....</b>	<b>147</b>
4.1	Testing C Applications in the PC Environment .....	147
4.2	Debugging C Applications in the PLC .....	150
4.2.1	Message Mode Debugging .....	150
4.2.2	Reference Table Monitoring .....	150
<b>Chapter 5:</b>	<b>Conversion Notes and Series 90 Compatibility ...</b>	<b>151</b>
5.1	Series 90 Compatibility Header Files (PLCC9070.h and PLCC9030.h).....	151
5.2	Writing Directly to Discrete Memory.....	152
5.3	PLC Target Library Function Compatibility Issues .....	153
5.4	Compatibility Issues with Retentive Global Variables .....	153
5.5	“int” Type Issues.....	154
5.6	“enum” Type Issues .....	154
5.7	Non-Standard C Library Functions.....	154
5.8	Entry Point.....	154
5.9	C Standalone Programs .....	154
5.10	Use of Input Parameters as Pointers to Discrete Memory Tables.....	155
<b>Chapter 6:</b>	<b>Installed Sample Blocks.....</b>	<b>156</b>
6.1	SampleProj1 .....	156
6.2	SampleProj2 .....	157
<b>Chapter 7:</b>	<b>Migration Notes for RSTi-EP Compatibility .....</b>	<b>158</b>
7.1	C Block Executable memory considerations .....	158
7.2	Notes on unaligned memory access in C Application .....	158
7.3	Limitations on Double/float discrete data type macros .....	159
7.4	Limitations on typecasting of double data type.....	160
7.5	Notes on double data type pointer assignment limitations .....	161
7.6	Notes on usage of “ / “ and ” %” operators in C Application.....	162
<b>Appendix A:</b>	<b>Target Library Functions .....</b>	<b>163</b>
A-1	Target Library Reference Memory Functions and Macros .....	163
A-2	Target Library Fault Table Functions, Structures and Constants.....	177
A-3	Target Library General Functions, Structures and Constants .....	184

A-4 Target Library VME Functions, Structures and Constants..... 200

A-5 Target Library Error Functions, Structures and Constants ..... 202

A-6 Target Library Utility Functions, Structures and Constants ..... 202

**Appendix B: C Run-Time Library Functions ..... 204**

**Appendix C: Diagnostics ..... 217**

C-1 Issue: Compiler issues the following warning when the EnableAnsi flag is used: 217

C-2 Issue: Compiler issues the following statement: warning: `HUGE\_VAL' redefined.218

C-3 Issue: Compiler issues the following error statement: undefined reference to `isascii' when the EnableAnsi flag is used. In addition, the C Block will not store to the PLC. ... 218

C-4 Issue: On some Windows 2000 PCs, the local DOS Box Environment "path" variable is not used, resulting in the compile process failing because the path to the compiler batch file is not found. .... 218

---

## Warning, Caution & Danger Notes as Used in this Publication

---



### Warning

Warning notices are used in this publication to emphasize that hazardous voltages, currents, temperatures, or other conditions that could cause personal injury to exist in this equipment or may be associated with its use.

In situations where inattention could cause either personal injury or damage to equipment, a Warning notice is used.



### Caution

Caution notices are used where equipment might be damaged if care is not taken.

---

**Notes:** *Notes merely call attention to information that is especially significant to understanding and operating the equipment.*

---

These instructions do not purport to cover all details or variations in equipment, nor to provide for every possible contingency to be met during installation, operation, and maintenance. The information is supplied for informational purposes only, and Emerson makes no warranty as to the accuracy of the information included herein. Changes, modifications, and/or improvements to equipment and specifications are made periodically and these changes may or may not be reflected herein. It is understood that Emerson may make changes, modifications, or improvements to the equipment referenced herein or to the document itself at any time. This document is intended for trained personnel familiar with the Emerson products referenced herein.

Emerson may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not provide any license whatsoever to any of these patents.

Emerson provides this document and the information included therein as-is and without warranty of any kind, expressed or implied, including but not limited to any implied statutory warranty of merchantability or fitness for particular purpose.

# Chapter 1: Introduction

This manual contains essential information about the construction of C applications for PACSystems control systems. It is written for the experienced programmer who is familiar with both the C programming language and with the operation of PACSystems control systems. For more information about PACSystems, refer to the list of documents at the end of this chapter.

The PACSystems C Programmer's Toolkit contains libraries, utilities, and documentation required to create C applications for the PACSystems control system. C blocks are constructed using the ANSI C programming language using text editing and toolkit applications on a personal computer. The C blocks are incorporated into a PACSystems application program through PAC® Machine Edition programming software. Using the programming software, C blocks can be called from ladder logic or invoked by an I/O, module or timed interrupt. In the programming software, use the Add C Block feature to insert C blocks.

The PACSystems CPU supports one type of C block, which has the capabilities of both the Series 90-70-type C blocks and C function blocks. The PACSystems CPUs and the PACSystems C Toolkit do not support Standalone C Programs, which is a feature of the Series 90-70.

A PACSystems C block is, by default, limited to 256Kbytes in size, provided there is sufficient PLC memory. Examples of calculations that might be performed in C blocks include:

- Ramp/soak profiling
- Lead/lag calculation
- Message generation
- Input selection
- Arithmetic operations
- PID
- Sorting, moving and copying data

## 1.1 Revisions in this Manual

Rev	Date	Description
G	Aug-2019	Updated with C Blocks support for RSTi-EP 32-bit PLC's
F	Oct-2017	Updated for 64-bit C Blocks and compatible operating systems.
E	Jan 2012	General Update
D	Jun 2008	Additions to Chapter 3: and Appendix A:.

## 1.2 Documentation

### PACSystems Manuals

PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual	GFK-2222
PACSystems RX7i, RX3i and RSTi-EP CPU Programmer's Reference Manual	GFK-2950
PACSystems RX7i, RX3i and RSTi-EP TCP/IP Ethernet Communications User Manual	GFK-2224
PACSystems Hot Standby CPU Redundancy User Manual	GFK-2308
PACSystems Battery and Energy Pack Manual	GFK-2741
PAC Machine Edition Logic Developer Getting Started	GFK-1918
Proficy Process Systems Getting Started Guide	GFK-2487
PACSystems RXi, RX3i, RX7i and RSTi-EP Controller Secure Deployment Guide	GFK-2830

### RX3i Manuals

PACSystems RX3i System Manual	GFK-2314
PACSystems RX3i PROFINET Controller Command Line Interface Manual	GFK-2572
PACSystems RX3i Max-On Hot Standby Redundancy User's Manual	GFK-2409
PACSystems RX3i PROFINET Scanner Manual	GFK-2737
PACSystems RX3i CEP PROFINET Scanner User Manual	GFK-2883
PACSystems HART Pass Through User Manual	GFK-2929
PACSystems RX3i PROFINET Scanner Important Product Information	GFK-2573

### VersaMax Manuals

VersaMax PROFINET Scanner Manual	GFK-2721
----------------------------------	----------

In addition to these manuals, datasheets and product update documents describe individual modules and product revisions. The most recent PACSystems documentation is available on the support website <https://www.emerson.com/Industrial-Automation-Controls/support>.



## Chapter 2: Installation

This chapter explains how to install the PACSystems C Toolkit software on your personal computer.

This chapter provides the following information:

- What you will need to use the C Toolkit software
- Installing the C Toolkit for PACSystems
- Running the C Toolkit
- C Toolkit file structure

### 2.1 System Requirements

To use the C Toolkit, you will need the following (at a minimum)<sup>1</sup>:

- PC: Pentium class processor, 166MHz or better
- RAM: 128MB, minimum
- Free Disk Space: 100MB, minimum
- Operating System:

Version	Operating Systems Supported:
Earlier C Toolkit Releases such as 7.00	Windows® XP Professional (service pack 1 recommended) or Windows® 2000 Professional (service pack 3 recommended)
C Toolkit Release 8.00 (64-bit support)	Windows® 7/10 Professional
C Toolkit Release 8.10 (RSTi-EP support)	Windows® 7/10 Professional

**Note:** *The C Programmer's Toolkit for PACSystems has not been qualified for use with the Windows Vista™ operating system*

### 2.2 Installing the C Toolkit for PACSystems

This section describes how to install the C Toolkit software for PACSystems on your computer and how to set up your computer to use the Toolkit

<sup>1</sup> More modern operating systems, such as Windows 10, specify PC and memory requirements which will exceed the indicated minima.

## ⚠ CAUTION

Before installing the C Toolkit to the same directory as a previous installation, you should first uninstall the previous version. Failure to do so may cause the Toolkit to function incorrectly.

### 2.2.1 To install the Toolkit

1. Execute the setup.exe file.
2. Click the Next button. The next installation screen displays the default location where the Toolkit will be installed: C:\Emerson Software\PACSystemsCToolkit.  
You can change the install directory either by entering a directory path or by browsing to the desired directory.
3. Click the Next button. The install program prompts you to create the installation directory if necessary. The install program then asks if you want to proceed with the installation in the designated directory.
4. To complete the installation, click the Start button. The install package installs the software and user documentation components in the designated directory. The installation program also installs an icon on your desktop.

When this operation is complete, the final installation screen is presented. This screen provides the option for viewing the readme.txt, which presents important start-up information.

Figure 1:



5. Click Next.
6. To exit the installation program, click the Exit button. This launches the Toolkit, which brings up a DOS box in the user project area. From the DOS box, you can navigate to your project directories and compile C files. The initial screen will be similar to the example shown below:

Figure 2:

```
PACSystems(TM) C Toolkit
Toolkit Release: 8.10
Toolkit Build Number: 10D1
Microsoft Windows [Version 10.0.18363.535]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\EmersonSoftware\PACSystemsCToolkit\Projects>
```

## 2.3 Running C Toolkit

To start the toolkit, double click the desktop icon (PACSystems(TM) C Toolkit) or use the Start menu to execute the file *ctkPACS.bat* located at the Toolkit's root directory.

In addition, you can also open an independent DOS window, navigate to the directory containing the *ctkPACS.bat* file, run the *ctkPACS.bat* file, navigate to your project and then compile the project.

Figure 3:


















Because the *ctkPACS.bat* file does not change the *autoexec.bat* file, the environment variables are only valid for the life of the DOS window. This means that you can run another version of the toolkit on the same machine without conflicts between the two packages because the environment variables are local to each DOS window.

## 2.4 C Toolkit File Structure

The file structure of the installed C Toolkit is shown below.

Figure 4:

	bin	File folder
	Compilers	File folder
	Docs	File folder
	Projects	File folder
	Targets	File folder
	ctkPACS.bat	Windows Batch File
	GNU.txt	Text Document
	index.htm	HTML Document
	license.txt	Text Document
	readme.txt	Text Document
	readmePACRSTi.txt	Text Document
	readmePACRX.txt	Text Document
	readmePACRX3i.txt	Text Document
	readmePACRX7i.txt	Text Document
	Uninstal.exe	Application

## 2.4.1 Directories

<b>Bin</b>	contains the binary executable files used by C Toolkit.
<b>Compilers</b>	contains the tools to compile and link your C Block file(s).
<b>Docs</b>	contains local copies of user documentation in a standard format (html or pdf). To navigate to the user documentation, double click the index.htm file located in the root directory. The index.htm file provides links to the documentation on the Support web site.
<b>Projects</b>	can be used to contain your C Block projects and in addition contains sample C Block projects.
<b>Targets</b>	contains a target subdirectory and a debug subdirectory for each supported target. The target subdirectories contain subdirectories for the C Run Time and Target Library header files and compilation programs specifically needed for compiling C Block files for that particular target. The debug subdirectories contain files needed to compile and debug C Blocks on the PC using the Cygwin development environment. The Targets directory also contains a Common Files subdirectory that contains files common to more than one target.

## 2.4.2 Files

<b>ctkPACS.bat</b>	opens a DOS box and sets up path and environment variables so that C Blocks can be compiled from any location on your computer.
<b>GNU.txt</b>	lists the locations of files covered by the GNU General Public License.
<b>index.htm</b>	contains links to the user documentation.
<b>license.txt</b>	contains the license information for the C Toolkit.
<b>readme.txt</b>	indicates how to get to the readme file for a particular target. This is the only file available in release 8.00 (64-bit).
<b>readmePACRX.txt</b>	contains start-up information for PACs targets.
<b>readmePACRX3i.txt</b>	contains start-up information for PAC RX3i targets.
<b>readmePACRX7i.txt</b>	contains start-up information for PAC RX7i targets.
<b>readmePACRSTi.txt</b>	contains start-up information for PAC RSTi-EP targets.
<b>uninstall.exe</b>	removes the C Toolkit from your computer. Your project directories are not removed during the uninstall process.

## 2.5 Uninstalling C Toolkit

To uninstall the C Toolkit, execute the Uninstal.exe file.

This deletes all files created by the C Toolkit install program. Any new files that you have created in the directory structure will remain as user project files.

## Chapter 3: Writing a C Application

This chapter contains information needed to write C applications for the PACSystems control system. It includes details on declaring parameters, accessing CPU reference memory, and using standard library routines.

• Name Requirements	page 7
• C Applications in the PACSystems Environment	8
• PACSystems C Block Structure	19
• PLC Reference Memory Access	23
• Standard Library Routines	32
• Application Considerations	131

---

**Note:** For information on testing and debugging C applications, refer to chapter 4. For information on compatibility with Series 90-70 and Series 90-30 C applications and issues to be aware of when converting C applications from 90-70 or 90-30 to PACSystems, refer to chapter 5

---

The C source code used to build C applications may be created using the text editor of your choice, provided that the output from your editor is compatible with the GNU C compiler. (Word processors are not recommended for editing C source code.) In addition, your editor must properly handle both DOS- and UNIX-type line feeds (Note that Notepad does not handle UNIX style line feeds and may not display some C Toolkit files correctly).

It is also recommended that each C application be developed in its own subdirectory. One approach would be to use the project subdirectory created when the C Toolkit was installed. As each application is developed, a new subdirectory under the \Projects\ subdirectory is created: for example,

Projects\Ramp

Projects\Limit

Projects\Press

etc.

---

**Note:** When migrating 32-bit C blocks to 64-bit C blocks, care must be taken. Consult coding practices for migrating C code from 32-bit to 64-bit.

---

### 3.1 Name Requirements

#### 3.1.1 File Names

The filename of a C Block (for example, myCBlock.gefElf) (i.e. the string that precedes the extension) must conform to Machine Edition block naming conventions (i.e. a maximum length of 31 characters, first character must be a letter, no spaces). In addition, do not use the filename "Rel". This name is reserved by the C Toolkit (see Section 3.2.3.4 Compiling User C Blocks Under an Older Toolkit Version).

---

**Note:** A compiled 32-bit C block has a .gefElf file extension, whereas a compiled 64-bit C block has a .gefElf64 file extension, .gefElfARM for 32-bit RSTi-EP. Examples in the text may use .gefElf, but .gefElf64 are substituted when working with 64 bit C Blocks.

---

## 3.1.2 Reserved Names

To avoid C Toolkit and user naming conflicts, you should not use any of the following types of names in your C Block application:

1. Names that begin with “GEF\_”
2. Names that begin with a period “.”. For example, “.mydata”

Failure to follow these rules could result in compilation or store errors and possibly incorrect operation.

## 3.2 C Applications in the PACSystems Environment

### 3.2.1 Developing a C Block

For PACSystems, there is only one type of C Block and this block can be re-entrant if re-entrant guidelines are followed. C Block source code is written using a text editor of choice (with restrictions as outlined at the beginning of this chapter). In order to use the Target Library functions and macros, you must use one of the following lines at top of the C file:

```
#include <PACRXPlc.h> /* For C blocks that run on any PACSystems PLC */  
#include <PACRX7iPlc.h> /* For C blocks that use features only available on an RX7i */  
#include <PACRX3iPlc.h> /* For C blocks that use features only available on an RX3i */  
#include <PACRSTiPlc.h> /* For C blocks that use features only available on an RSTi-EP */
```

---

**Note:** In the 90-70 there are two types of C blocks (C BLK & C FBK). The C BLK type cannot be re-entrant but can make use of the C Run-Time library. The C FBK can be re-entrant but cannot use the C Run-Time library.

---

A list of the Target Library functions and macros are listed in Appendix A.

To use the C Run-Time Library functions, you must include one or more of the following files as appropriate at the top of the C file:

```
#include <stdio.h> /* Input/Output */  
#include <math.h> /* Math */  
#include <stdlib.h> /* Math, Data Conversion, Search */  
#include <string.h> /* String Manipulation, Internationalization */  
#include <time.h> /* Time */  
#include <ctype.h> /* Character Classification and Conversion */
```

A list of the C Run-Time library functions supported by the PACSystems is provided in Appendix B.

The paths to these include files are set up when the C compiler runs, therefore the full paths are not required in the “include” file names. After including the appropriate header files, you can write a C block, using library calls as needed to implement the desired functionality. The C Block file or set of C Block files must have one and only one function titled “GefMain” to act as the entry point. A brief example is shown in Figure 5.

**Figure 5: Example C Block Source File**

```
/* myCFile.c */
#include <stdio.h>
#include <PACRXPlc.h>
T_INT32 status;
T_INT32 status2 = 1;
T_INT32 failCount = 0;
T_INT32 GefMain(T_INT16 *x1, T_INT16 *y1)
{
    if (*x1 != 0)
    {
        RW(10) = *x1; /*write x1 to %R10 as word */
        return GEF_EXECUTION_OK;
    }
    else
    {
        status = GEF_EXECUTION_ERROR;
        status2 = failCount;
        failCount++;
        return status;
    }
}
```

The input parameters to the main block (x1 and y1) are derived from the input/output parameters in the ladder program that calls the C Block. Input parameters are always passed as pointers. An example is shown below:

**Figure 6: Invoking a C block from Ladder Program**



For this example, x1 points to the memory location of %R1 and y1 points to the memory location of %R2. A return value of GEF\_EXECUTION\_OK enables power flow output from the C Block while a return value of GEF\_EXECUTION\_ERROR results in no power flow from the output of the C Block.

## 3.2.2 C Toolkit Variable Types

To maintain portability and reduce errors, it is recommended that you use the basic types defined by the header file `ctkGefTypes.h` and the files it includes. This file is located in the Toolkit subdirectory `Targets\CommonFiles\IncCommon`. This file defines the recommended basic signed and unsigned types from 8- or 64-bit quantities. These types are described below:

**Table 1. Variable Types**

C Toolkit Variable Types	Description	Corresponding Programmer Variable Type	Notes
T_BOOLEAN	8-bit type where 0 means FALSE and non-zero means TRUE. However TRUE typically is set to a value of 0x01	BOOL	In the programmer/PLC, this type represents a single bit.  <i>Note: when passing a Boolean parameter to a C block, the memory address of the PLC reference table memory must be byte-aligned because the C Block is passed a pointer to a Byte of reference memory. The C user must then mask off and test the least significant bit to get the Boolean state.</i>
T_BYTE	8-bit unsigned type.	BYTE	
T_WORD	16-bit unsigned type	WORD	
T_DWORD	32-bit unsigned type	DWORD	
T_INT8	8-bit signed type	NA	
T_INT16	16-bit signed type	INT	<b>⚠ CAUTION</b> Using “int” in the C source results in a 32-bit signed type that does not properly match the programmer’s “INT” type.
T_INT32	32-bit signed type	DINT	
T_UINT8	8-bit unsigned type	BYTE	



C Toolkit Variable Types	Description	Corresponding Programmer Variable Type	Notes
T_UINT16	16-bit unsigned type	UINT	
T_UINT32	32-bit unsigned type	DWORD	
T_UINT64	64-bit unsigned type	NA	
T_REAL32	32-bit floating point type	REAL	This is equivalent to “float.”
T_REAL64	64-bit floating point type	LREAL	This is equivalent to “double.”

**Table 2. Standard Basic Types Commonly Used for C Block Applications**

C Toolkit Variable Types	Description	Corresponding Programmer Variable Type	Notes
char	8-bit character	NA	Similar to a BYTE in programmer.
double	64-bit floating point	LREAL	

If you include the header file PLCC9070.h or PLCC9030.h, it equates Series 90 C Toolkit basic types and the corresponding PACSystems C Toolkit basic types. This is shown in the following table:

**Table 3. Relationship Between Series 90 and PACSystems Basic Types**

90-30/90-70 Variable type	Corresponding PACSystems C Toolkit Variable Type
byte	T_BYTE
word	T_WORD
dword	T_DWORD
dint	T_INT32
bflow	T_BOOLEAN

### 3.2.3 Compiling

After developing a C Block as described in Section 3.2.1, Developing a C Block, the C Block must be compiled to create a relocatable object file that can be stored into the PLC.

#### Compiling a Single C File

To compile the C Block:

1. Start the C Toolkit by double clicking on the PACSystems C Toolkit icon on your desktop, double clicking on the ctkPACS.bat file through Windows explorer or using the Start->Programs menu.
2. In the C Toolkit DOS box, navigate to the project directory containing the C block file.
3. Type the appropriate compile command, followed by your file name.
  - To compile a C Block that can be run on any PACSystems RX PLC, use the command: **compileCPACRX <file name>**. This is the only command available for 64-bit C blocks.
  - To compile a C Block that uses functionality that is available only on an RX3i, use the command: **compileCPACRX3i <file name>**. Compatible with 32-bit C Blocks only.
  - To compile a C Block that uses functionality that is available only on an RX7i, use the command: **compileCPACRX7i <file name>**. Compatible with 32-bit C Blocks only.

- To compile a C Block that uses functionality that is available only on an RSTi-EP, use the command: **compileCPACRSTi <file name>**. This is the only command available for 32-bit RSTi-EP platform.

For example, to run the RX7i compiler for a C file called “myCFile”, type:

**compileCPACRX7i myCFile**

If there are errors or warnings, they are noted on the screen. If the compile is successful (no errors), an output file is produced with the same base name as the input file and the extension “.gefElf”. The file is placed in a subdirectory under your project directory called “plc” so that it is clear which file is intended for downloading to the PLC. For the “myCFile” example, the following file is produced:

**myCFile.gefElf**

myCFile.gefElf contains the compiled relocate-able code that is used by the PLC to load the C Block into user memory.

See section 3.6.17, Restricting Compilation to a Specific Target if you want your C Block to always be compiled for a specific target.

## Compiling Multiple C Files

If you want to have multiple C files compiled and linked together, you need to create a file called “sources” and include a line that specifies the files to compile. This line must start with the word “CFILENAMES=” (all capitals, no spaces) followed by the filenames (there can be multiple spaces or tabs between “CFILENAMES=” and the first file and multiple spaces or tabs between each filename). An example of this line is shown below:

**CFILENAMES= myCFile1.c myCFile2.c myCFile3.c**

If the list of files is long, a continuation symbol “\” may be added to improve readability in the file as shown below:

**CFILENAMES= myCFile1.c myCFile2.c \  
myCFile3.c**

The “sources” file must be in the same project directory as the other C source files when compiling.

- To compile multiple C files into a C Block that can be run on any PACSystems RX PLC, use the command: **compileCPACRX**.
- To compile multiple C files into a C Block that uses functionality that is available only on an RX3i, use the command: **compileCPACRX3i**.
- To compile multiple C files into a C Block that uses functionality that is available only on an RX7i, use the command: **compileCPACRX7i**.
- To compile multiple C files into a C Block that uses functionality that is available only on an RSTi-EP, use the command: **compileCPACRSTi**.

For example, to compile multiple C files for a C Block that can be run on any PACSystems RX PLC target, enter:

### compileCPACRX

In this case, a file name is not given because the file name set comes from the “sources” file. The name of the output file is the base name of the first file in the sources file list plus the “.gefElf” or “.gefElf64” extension. For the example given above, the output file is: **myCFile1.gefElf**.

Again, this file will be located in the subdirectory “plc”. When working with multiple files, you will need to add the keyword **extern** to any function or global variable that is referenced and declared in another file. For example, if myCFile1 uses myFunction2 and myVar2 in myCFile2, myCFile1 must declare these “extern” as shown below:

```
extern int myVar2;
```

```
extern void myFunction2(void);
```

## Specifying Compiler Options

You can specify the following compiler options by supplying keywords after the filename for the single file case or setting flag1 and flag2 with one of the keywords in the sources file when compiling multiple files:

1. Disable Stack Checking (Keyword = DisableStackCheck): this disables stack checking on every user function call. This decreases C Block execution time but eliminates a check to determine if a particular function call will overrun the user program stack which could lead to data corruption and user program failure.
2. Enable ANSI compatibility (Keyword = EnableAnsi): this causes the compiler to enforce ANSI standards such as the prevention of the use of the double slash for comments.

An example of a single file compile using these keywords is shown below:

```
compileCPACRX myCFile DisableStackCheck EnableAnsi
```

An example of a multiple file compile using these keywords is shown below. In a file with the name “sources” include the following lines:

```
CFILENAMES= myCFile1.c      myCFile2.c      myCFile3.c
```

```
flag1 = DisableStackCheck
```

```
flag2 = EnableAnsi
```

To compile, type the following line in the DOS box in the same directory as the “sources” file:

```
CompileCPACRX
```

You can also link pre-compiled object files by using the following line in the “sources” file:

```
OFILENAMES=myCFile4.plcO
```

Multiple object files can be linked by placing space (spaces or TABS) between file names. In addition, the file names can be on separate lines if the continuation slash is added at the end of the line as shown below:

```
OFILENAMES= myCFile4.plcO      myCFile5.plcO \  
              MyCFile6.plcO
```

The following lines show an example of a “sources” file that compiles multiple C source files, multiple object files and sets compile flags:

```
CFILENAMES= myCFile1.c      myCFile2.c      myCFile3.c  
OFILENAMES=myCFile4.plcO    myCFile5.plcO  
flag1 = DisableStackCheck  
flag2 = EnableAnsi
```

PLC object files can be created by using the flag DisableGefLibLink. To create myCFile4.plcO in the current directory, type the following line:

```
compileCPACRX myCFile4 DisableGefLibLink
```

## Compiling User C Blocks Under an Older Toolkit Version

If you are developing C blocks for a PLC with an older firmware version, the C Toolkit allows the code to be compiled under the limitations of an older C Toolkit version. You can specify the Toolkit release on the command line (as the last two parameters) at the time the C block is compiled. If a version is not specified, the C code will be compiled with the most recent version (newest feature set). For example:

Normal command:

```
compileCPACRX myCFile OR compileCPACRX (assumes a “sources” file)
```

Release-specifying command example:

```
compileCPACRX myFile Rel 1_0 OR compileCPACRX Rel 1_0 (assumes a “sources” file)
```

In this example, the release specified in the second command is 1.0. Release numbers should be preceded by the keyword “Rel” so that the compile batch file knows that “compileCPACRX Rel 1\_0” is meant to compile the C code specified in a sources file within the constraints of release 1.0 of the C Toolkit. (The name of the file containing the user’s C code, if specified on the command line, cannot be “Rel.”)

As of Release 5.00, the following revisions can be specified on the command line after the keyword “Rel”:

```
1_0  
1_5  
2_0  
2_5  
3_5  
5_0
```

## 3.2.4 Associating a Compiled C Block with the Application Program

After the program is compiled, you must associate the \*.gefElf file with a C Block in your PLC program using the programmer. The C Block must have the same number of parameters as the GefMain function's input parameter signature as illustrated in Figure 5: Example C Block Source File. However, there is not a check to determine if the signatures match. In cases where the signatures do not match, the C Block may not behave correctly.

## 3.2.5 Adding Blocks through the Machine Edition Programmer

Before importing the block into Machine Edition, the C application source file must be compiled and linked to create the relocate-able version of the C application (\*.gefElf).

Once the relocate-able version of a C application source file is created, the file needs to be added to a target within your CME project as follows:

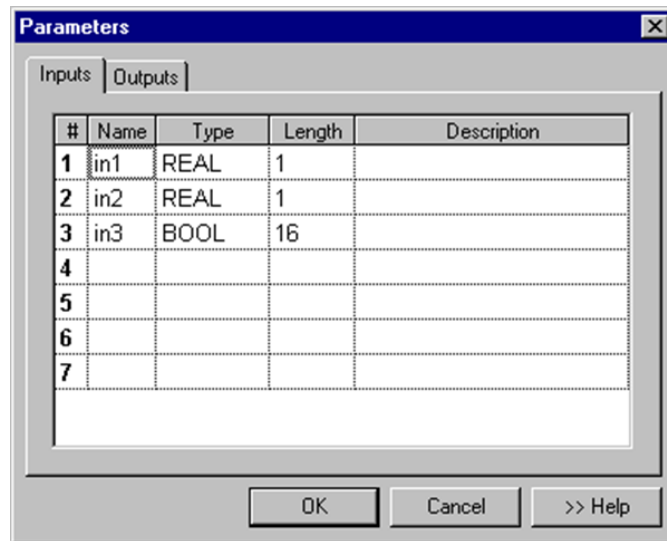
1. In the Project tab, expand the Logic node.
2. Right click the Program Block node under the Logic node.
3. Select Add C Block. This brings up a file navigation dialog box.
4. Navigate to the \*.gefElf file and click the Open button to add the C Block to the folder.

## 3.2.6 Specifying Parameters

To specify the parameters for a C Block with one or more input/output parameters, click on the C Block. In the properties page for the C Block, click the Parameters item and then click on the button provided. This opens the Parameters dialog box containing two tabs, one for inputs and one for outputs. For each input/output, provide:

- Name
- Type (BOOL, BYTE, DINT, DWORD, INT, LREAL, REAL, UINT, WORD) (See Section 3.2.2, C Toolkit Variable Types for information on how to map programmer/PLC types to C Toolkit types.)
- Length

**Figure 7: C Block Parameters in Properties Page**



**Note:** All parameters must be declared, even if some of them are NULL. (A NULL parameter may be used when converting a 90-70 C Block to PACSystems.) To declare the parameter in Machine Edition, the parameter must have both a type and a length. If the type is specified as NONE, with no length, Machine Edition does not generate the parameter.

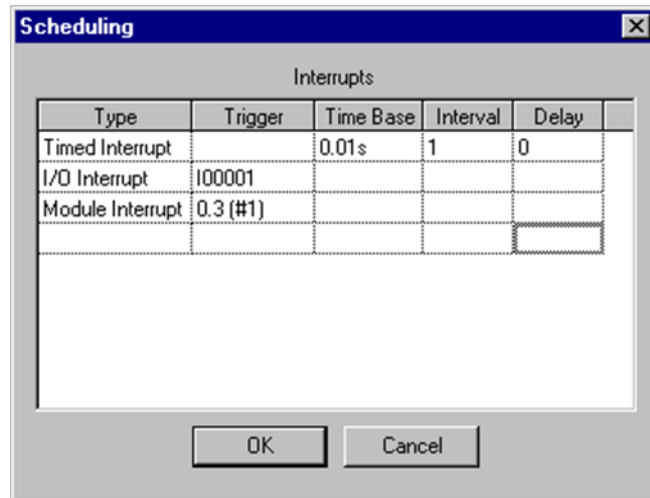
## 3.2.7 Scheduling C Blocks

To schedule a C Block as a timed, I/O, or module Interrupt, click on the C Block. In the Properties page for the C Block, click the Scheduling item and then click on the button provided. This opens the Scheduling dialog box that allows you to select:

- Type: Timed, I/O, or Module Interrupt
- Trigger: I/O address for I/O or Module Interrupt
- Time Base: 0.01001s, 0.01s, 0.1s, or 1s base for timed interrupts
- Interval: the number of time base units between timed interrupts
- Delay: initial delay before the timer starts for timed interrupts

Please note that only C blocks with no Input and Output parameters may be scheduled.

Figure 8: Scheduling a C Block

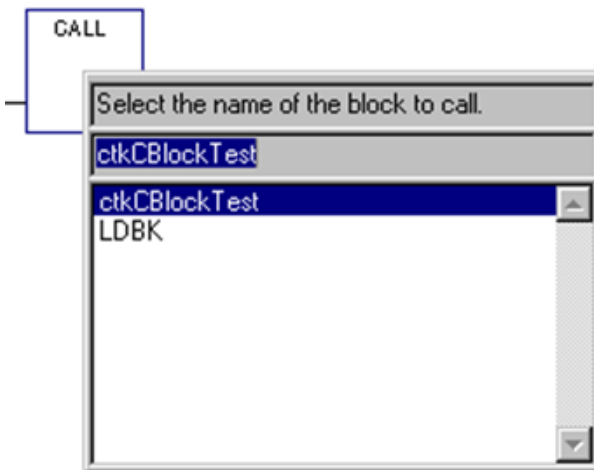


### 3.2.8 Using a C Block in an LD or FBD Program

To use a C Block in the ladder or function block diagram program, place a Call instruction in the desired location. Select the C block desired. If the block has parameters, provide reference memory locations for each input and output parameter.

To use a C block in an ST program, see below.

Figure 9: Calling a C Block from Program



### 3.2.9 Using a C Block in an ST Program

You can call a C block from an ST program by using a Block Call statement. A block call to a parameterized C block can use either the informal or formal convention.



Call to an unparameterized C block:

**My\_C\_Block;**

Call to a parameterized block using the informal convention:

**My\_C\_Block(my\_Input1, my\_Input2, my\_Output2, my\_Output1);**

Call to a parameterized block using the formal convention (parameters can be in any order):

**My\_C\_Block(Out1 => my\_Output1, In1 := my\_Input1, In2 := my\_Input2, Out2 => my\_Output2);**

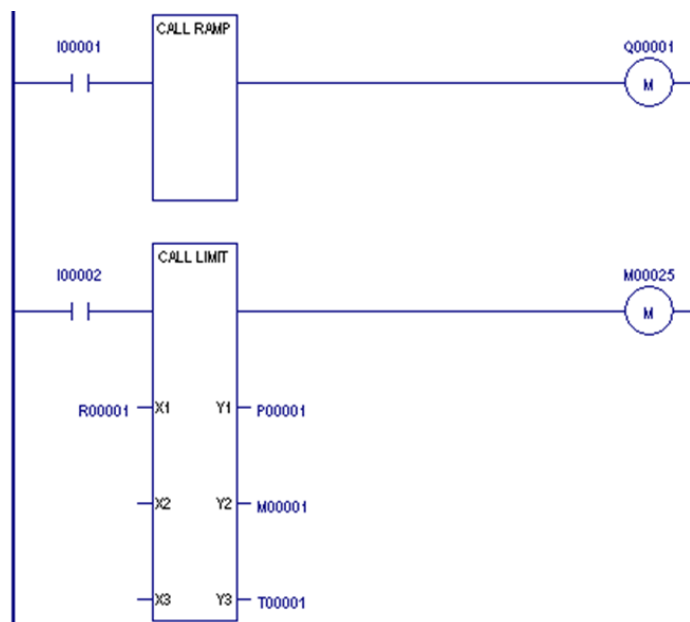
## 3.3 PACSystems C Block Structure

A C block can be invoked in one of five ways:

1. As a sub-block of the main block.
2. As a sub-block of an LD, ST, or FBD block.
3. As a sub-block of an LD, ST, or FBD block with parameters (parameterized block).
4. As an I/O, timed, or module interrupt block.
5. As a sub-block of an interrupt block.

Blocks invoked as a sub-block of main, or as a sub-block of an interrupt block may have up to sixty-three input and sixty-four output parameters. The input parameters do not have to be paired with output parameters as required in the Series 90-70. Blocks invoked as an I/O, timed, or module interrupt cannot have parameters. Shown below are two ladder logic rungs containing a C block with zero parameters and a C Block with three input and three output parameters.

**Figure 10: Ladder Logic Calls to C Blocks**



---

**Note:** *The Enable output (ENO) is present regardless of whether the block has parameters and is set based on the function return result (either GEF\_EXECUTION\_OK or GEF\_EXECUTION\_ERROR). Each block is written as a separate application that is linked and located during the program store process.*

---

Appropriate definitions of GEF\_EXECUTION\_OK or GEF\_EXECUTION\_ERROR are given in the ctk.h file, which is included by the header file PACRXPlc.h. The ctk.h file is located in the subdirectory PACSystems CToolkit\Targets\CommonFiles\IncCommon. The gefElf file produced by the build process of a block must be added to the program folder via CME using the Add C Block command.

The main function in each block must always be called GefMain. Any legal C declaration and code may be used in a C block. The file PACRXPlc.h, installed as part of the C Toolkit, should be included in the block source file(s). PACRXPlc.h contains or includes other files that contain declarations, definitions, and macros used in writing blocks.

The following example shows the basic components of a block with no parameters:

```
#include PACRXPLC.h /* PACSystems RX interface file */

int GefMain ()
{
    /*value of function block ENO output determined by return value */
    return GEF_EXECUTION_OK;
}
```

### 3.3.1 Variable Declarations

Global and static variables may be used in a C block. The space allocated for them is taken from the 256K byte default space allowed for each block. Local, or automatic, variables are allocated on the stack. PACSystems guarantees that a minimum of 5120 bytes is available on the stack before calling a C block. If this amount of space is not available before calling the block, a diagnostic application fault will be logged in the fault table.

### 3.3.2 Stack Overflow Checking

Stack overflow checking is enabled by default.

If C block stack checking is enabled when the block is built and the CPU detects that there is not enough space available on the stack when calling a user function within a block, an application fault will be logged in the controller fault table and the block will be exited at the point where the potential stack overflow is detected. The block ENO output will be turned off. To resolve the problem, you will need to evaluate if there is a problem in your application, such as a recursion (a block calling itself) or increase the stack size. Stack size can be increased in 8K byte increments on the \_MAIN Block properties page in the programmer.

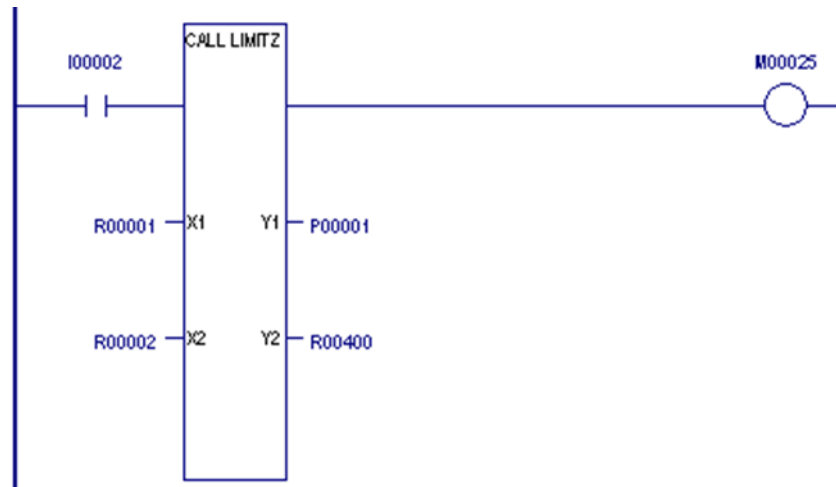
If C block stack checking is enabled when the block is built and the CPU detects the stack has already overflowed when calling a user function within a C block, a fatal application fault will be logged in the controller fault table and the PLC will be placed in Stop Faulted

mode. In some cases, such as when a function allocates a large amount of local or automatic variables in the stack, and the stack depth is near the bottom of the stack, a page fault may occur, and the CPU will be placed in CPU halted mode.

If stack checking is disabled via the block build process and the application exceeds the allocated stack space, a page fault may occur, or the CPU may receive invalid data.

The order of the parameter declarations must match the CALL instruction parameter order, with the input parameters followed by the output parameters. The declaration code shown below could be used for a block that has two input and two output parameters.

**Figure 11: Matching Parameters Between Call and C Block**

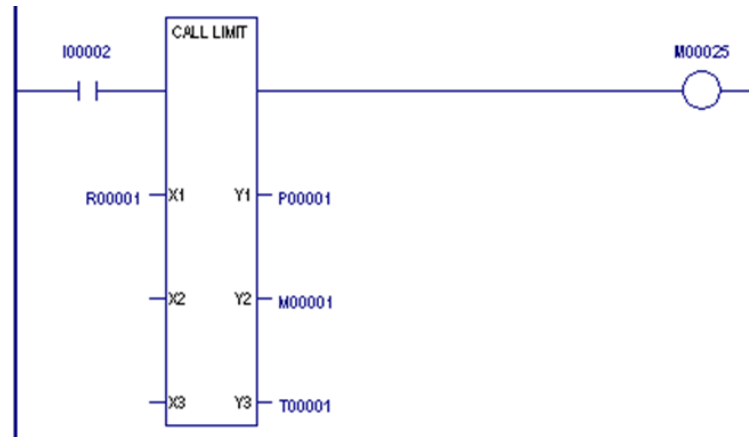


```
int GefMain (X1, X2, Y1, Y2)
/*X1 - pointer to a single 16-bit integer */
T_INT16 *X1;
/*X2 - pointer to a 256-element array of integers */
T_INT16 X2[256];

/*Y1 - pointer to a structure containing a 16-bit integer */
/* and a floating-point variable */
struct
{
T_INT16 a;
T_REAL32 b;
} *Y1;
/*Y2 - pointer to an unsigned 16-bit integer */
T_WORD *Y2;
/* Body of GefMain function starts here */
```

It is not required that all of the CALL instruction parameters be used. If a CALL instruction parameter is not used, a NULL pointer is passed as that parameter's value. The parameter must still be declared for the C Block in the programmer, so that subsequent parameters are lined up correctly with their pointers. In the following example, a NULL pointer is passed in for the second and third input parameters.

**Figure 12: Reserving Space for Unused Parameters to a C Block**



```
int GefMain(x1, x2, x3, y1, y2, y3)
{
    T_INT16 *x1;
    T_INT16 *x2;    /* placeholder for unused parameter, value is null */
    T_INT16 *x3;    /* placeholder for unused parameter, value is null */
    T_INT16 *y1;
    T_INT16 *y2;
    T_INT16 *y3;

    {
        *y1 = *x1; /* Copy value at x1 to y1 */
        *y2 = *x1 * 2; /* copy twice the value at x1 to y2 */
        *y3 = *x1 * 3; /* Copy three times the value at x1 to y3 */
        return(GEF_EXECUTION_OK)
    }
}
```

### 3.3.3 Parameter Pointer Validation

The ladder logic program provides pointers to the variables that are passed into the block's **GefMain ( )** function. Since it is not required to provide variables for all input/output parameters, you should check to make sure a pointer is not NULL before using it in your application. An example of this NULL pointer checking is shown below:

```
int GefMain (T_INT16 *x1, T_INT16 *x2, T_INT16 *x3, T_INT16 *y2, T_INT16 *y3)
{
    /* Ensure that required parameters were provided by caller */
    if ((x1==NULL) || (x2==NULL) || (x3==NULL) || (y1==NULL) || (y2==NULL) || (y3==NULL))
        return(GEF_EXECUTION_ERROR);

    /* Required parameters are present. */
    *y1 = *x1; /* Copy value at x1 to y1 */
    *y2 = *x1 * 2; /* copy twice the value at x1 to y2 */
    *y3 = *x1 * 3; /* Copy three times the value at x1 to y3 */
    return(GEF_EXECUTION_OK)
}
```

## 3.4 PLC Reference Memory Access

PACSystems reference address and diagnostic memory may be read and written directly via macros defined in **ctkRefMem.h**, which is included with PACRx/PACRSTi. Most of these macros consist of a string of capitalized letters for non-discrete memory and Title Case for discrete memory, which indicate the PACSystems reference type (and in some cases, the type of operation to be performed) followed by the reference offset in parentheses. In general, PLC reference memories may be accessed via these macros as bits, bytes (8-bit values), words (16-bit values), double words (32-bit values), single precision floating point numbers (32 bits), or double precision floating point numbers (64 bits).

### ⚠ CAUTION

Use extreme caution with the following discrete macros. These macros directly access discrete memory without taking into account corresponding override and transition memory.

%lb (x)	%lw (x)	%li (x)	%ld (x)
%Qb (x)	%Qw (x)	%Qi (x)	%Qd (x)
%Mb (x)	%Mw (x)	%Mi (x)	%Md (x)
%Tb (x)	%Tw (x)	%Ti (x)	%Td (x)
%Gb (x)	%Gw (x)	%Gi (x)	%Gd (x)
%Sb (x)	%Sw (x)	%Si (x)	%Sd (x)
%SAb (x)	%SAw (x)	%SAi (x)	%SAd (x)
%SBb (x)	%SBw (x)	%SBI (x)	%SBd (x)

%SCb (x)	%SCw (x)	%SCi (x)	%SCd (x)
----------	----------	----------	----------

**Note:** This behavior is different from the Series 90-70 and Series 90-30 C feature

### Potential consequences:

- Inputs, outputs or internal discrete memory (for example %M) that are overridden (forced) to a particular state can change to the opposite of the overridden state if a write operation is performed using these macros.
- Transitions on discrete memory will not be detected, potentially affecting transition sensitive logic.

### Alternatives:

- Use the following functions to write to discrete memory: WritePLCByte, WritePLCWord, WritePLCINT, WritePLCDint, PLMemCopy, SetBit, ClearBit, WritePLCDouble.

The complete set of reference type designators are as follows:

Reference Type	Description
%I	Discrete input references (use only for reading reference memory)
%Q	Discrete output references (use only for reading reference memory)
%M	Discrete internal references (use only for reading reference memory)
%T	Discrete temporary references (use only for reading reference memory)
%G	Discrete global data references (use only for reading reference memory)
%S	Discrete system references (use only for reading reference memory)
%SA	Discrete maskable fault references
%SB	Discrete non-maskable fault references
%SC	Discrete fault summary references
%AI	Analog input registers
%AQ	Analog output registers
%R	System register references
%W	Bulk memory references
%P	Program registers (use to store program data from main)
%L	Local registers (use to store program data unique to a block)

## 3.4.1 How to Format a PLC Reference Access Macro

The table shown below gives the modifiers used with the PLC reference macros (listed in Appendix A). The format for usage of these macros is as follows:

The letter of reference type, followed by one of the modifiers followed by a parenthetical number for the address you wish to access; e.g.,

**RI(1)=3;**                      This assigns the integer value 3 to %R00001

**RW(2)=0x55AA;**              This assigns the word value 55AAh to %R00002

The data type modifiers are as follows:

Modifier	Description
B	Unsigned byte reference (8 bits, 0 -> 255)
W	Word reference (16 bits, 0 -> 65535)
I	Integer reference (signed 16 bits, -32768 -> 32767)
D	Double precision integer reference (signed 32 bits, -2147483648 -> 2147483647)
F	Floating point reference (32-bit IEEE floating point format)
Dbl	Double precision floating point reference (64-bit IEEE floating point format)

Certain combinations of reference type designators and data type modifiers are not supported. Those combinations that are supported have macros defined in the **ctkRefMem.h** file. Refer to Appendix A:, Section A-1 for the complete set of macros provided.

Macros that permit access to reference memories as bits are slightly different from macros that access the same reference memories as bytes, words, double words, and/or floating-point numbers. Bit access macros, byte access macros, word/integer access macros, word-memories-as-bytes access macros, and double word/floating point access macros are described on the following pages of this chapter.

## 3.4.2 Bit Macros

There are three bit macros defined for each reference memory type:

Macro	Description
BIT_TST_X	Tests the specified bit
BIT_SET_X	Sets the specified bit
BIT_CLR_X	Clears the specified bit

References in a C application to %I would use **BIT\_TST\_I()**, **BIT\_CLR\_I()**, or **BIT\_SET\_I()**. The macro name indicates that %I reference memory is to be operated on and the operation is tested (TST), cleared (CLR), or set (SET). The value contained in parentheses is the reference number of the item to be tested, cleared, or set (for example, 120 for %I120). The bit set and bit clear macros are separate C application source statements.

---

**Note:** The bit test macros return a Boolean value contained in a byte. The accessed bit is right justified (least significant bit) in the byte, that is, each of the bit test macros will evaluate to 0 if the bit is OFF or 1 if the bit is ON.

---

The C application shown below will set %Q137, %M29, and %T99 if %I120 is ON and will clear %Q137, %M29, and %T99 if %I120 is OFF:

Example:

```
#include "PACRXPlc.h"

int GeFMain() {
    if (BIT_TST_I(120)) {
        BIT_SET_Q(137);
        BIT_SET_M(29);
        BIT_SET_T(99);
    } else {
        BIT_CLR_Q(137);

        BIT_CLR_M(29);
        BIT_CLR_T(99);
    }
    return(GEF_EXECUTION_OK);
}
```

The bit macros for accessing word-oriented PLC memories (%R, %W, %P, %L, %AI, and %AQ) as bits are similar to the above description except that these macros require one additional parameter, namely, the position within the word of the bit being accessed. The three forms of bit macros for accessing word-oriented PLC memory are BIT\_SET\_, BIT\_CLR\_, and BIT\_TST\_ (to specify the type of operation) followed by R, W, P, L, AI, or AQ (to specify the PLC reference memory to be used). There are two required parameters to these macros:

1. The word in the reference memory to access (1 to highest reference available in the specified PLC memory).
2. The bit in the selected word to use (bit numbers 1 to 16, with bit 1 being the least significant or rightmost bit).

To illustrate the bit macros for word-oriented memory, consider the following section of a C application:

```
if (BIT_TST_R(135, 6) )
    BIT_SET_P(13, 4);
else
    BIT_CLR_AI(2,1);
```

This portion of a C application checks the sixth bit in %R135. If the bit is on (1), then the fourth bit in %P13 is to be set ON (1); otherwise, the first bit in %AI2 is to be set OFF (0).

---

**Note:** The "BIT\_" macros used to access bits in word-oriented memories use a 1- to 16-bit numbering scheme, with bit 1 being the least significant bit and bit 16 being the most significant bit.

---



## 3.4.3 Byte Macros

Macros are provided to read the PLC bit memories as bytes. These macros are `lb(x)`, `Qb(x)`, `Mb(x)`, `Tb(x)`, `Gb(x)`, `Sb(x)`, `SAb(x)`, `SBb(x)`, and `SCb(x)`.

### ⚠ CAUTION

Use extreme caution with the following discrete macros. These macros should not be used to write directly to discrete memory because they do not take into account corresponding override and transition memory. For details, refer to Section 3.4, PLC Reference Memory Access.

The parameter `x` in each of these macros should be replaced with the reference address of a bit which is contained in the byte; for example, if the byte containing `%M123` is needed, use `Mb(123)`. The byte access macros should only be used on the right-hand side of a C statement (read operation only).

The example that follows sets the variable `MyVar` equal to the byte starting at `%Q65` and ending at `%Q72`.

```
Example:
#include "PACRxPlc.h"

int GefMain() {
    T_BYTE MyVar;
    MyVar = Qb(72);

    return(GEF_EXECUTION_OK);
}
```

Accessing bytes from word-oriented memories (`%R`, `%W`, `%P`, `%L`, `%AQ`, and `%AI`) requires an additional parameter to indicate which byte is to be read or written. The symbols **HIBYTE** and **LOBYTE** are defined in **PACRxPLC.h** for this purpose. For example, your C application requires that the low byte of `%R5` be read into a C application local variable and then copied into the high byte of `%R17`:

```
Example:
#include "PACRxPLC.h"

int GefMain() {
    T_BYTE abyvar;

    abyvar = RB(5,LOBYTE); /* read low byte of %R5 */
    RB(17,HIBYTE) = abyvar; /* write high byte of %R17 */

    return(GEF_EXECUTION_OK);
}
```

## 3.4.4 Integer/Word Macros

All PLC reference memories may be accessed as 16-bit 2's complement integers (T\_INT16) or as 16-bit unsigned integers (T\_WORD). As an example, a C application needs to read %R123 as an unsigned 16-bit integer and write %P13 as a 2's complement 16-bit integer and store the values in separate local C source variables:

Example:

```
#include "PACRxPLC.h"

int GefMain () {
    T_WORD      word_val;
    T_INT16      int_val = -133;

    word_val = RW(123); /* read %R123 as a word */
    PI(13) = int_val; /* copy 2's complement integer to %P00013 */
    .
    return(GEF_EXECUTION_OK);
}
```

## 3.4.5 Double Word/Floating Point Macros

All PLC reference memories may be accessed as 32-bit signed integers (T\_INT32), but only the word-oriented memories (%R, %W, %P, %L, %AQ, and %AI) may be accessed as 32-bit floating point numbers (T\_REAL32). As an example, a C application needs to read %R77 as a 32-bit integer and write a single precision floating point value to %P6.

Example:

```
#include "PACRXPlc.h"

GefMain() {
    T_INT32      T_INT32_val;
    T_REAL32      fp_val = 15.56;

    INT21_val = RD(77); /* read %R77 as a 32-bit integer */
    PF(6) = fp_val; /* write %P6 as single precision floating point */
    return(GEF_EXECUTION_OK);
}
```

## 3.4.6 Double Precision Floating Point Macros

Word-oriented PLC reference memories (%AI, %AQ, %L, %P, %R, %W) may be accessed as 64-bit floating point values (T\_REAL64). As an example, a C application needs to read the LREAL variable in %R101 and write that value to the LREAL variable at %W50.

Example:

```
#include "PACRXPlc.h"
GefMain() {
    T_REAL64 lreal_value;
    lreal_value = RDbI(101);
    WDbI(50) = lreal_value;
}
```

## 3.4.7 Reference Memory Size Macros

Macros are defined in **ctkRefMem.h** for determining the size of each memory type. These macros are in the form X\_SIZE, where X is the memory type letter I, Q, M, T, G, S, R, W, AI, AQ, P, or L. Each of these size macros returns an unsigned integer value equal to the highest reference available in the specified reference memory. If the last available reference in the %I table is %I32768, when a C application uses the I\_SIZE macro, the value 32768 will be returned.

### CAUTION

The reference memory size macros should be used to determine the size of the memory types written within a C application. Reads and writes outside of the configured range can result in incorrect data or PLC CPU failure. A safer alternative is to use read/write PLC functions that perform address boundary checking. These functions are: WritePlcByte, WritePlcWord, WritePlcInt, WritePlcDint, PlcMemCopy, SetBit, ClearBit, ReadPlcByte, ReadPlcWord, ReadPlcInt, ReadPlcDint.

For example, a C application is created that takes an index as a single input parameter into the register table. The application is designed to index into the register table using the input parameter and copy the located value to the single output location (MOVE from source array registers [input parameter] to output parameter). This C application is to be designed so that it may be run on any PACSystems CPU, regardless of differing register memory table sizes:

```

Example:
#include "PACRxPlc.h"

int GefMain(T_WORD *X1, T_INT16 *Y1) {
    if ((X1 != NULL) && (Y1 != NULL)) {
        if (*X1 > R_SIZE) {
            /* Index into registers is too large! */
            return(ERROR);
        } else {
            /* Index into registers and copy value to output parameter*/
            *Y1 = RI(*X1);
        }

        return(GEF_EXECUTION_OK);
    }
    else return (GEF_EXECUTION_ERROR);
}

```

### 3.4.8 Transition, Alarm, and Fault Macros

Transition, alarm, and fault bits associated with reference memory can also be referenced. In addition, the special system %S contacts **FST\_SCN**, **LST\_SCN**, **T\_10MS**, **T\_100MS**, **T\_SEC**, **T\_MIN**, **ALW\_ON**, **ALW\_OFF**, **SY\_FULL**, and **IO\_FULL** are supported for C blocks.

The **FST\_EXE** macro is supported. This is high (1) the first time a block is executed. C Blocks and Parameterized Blocks inherit **FST\_EXE** from the calling block. Interrupt blocks (C, LD, FBD or ST) inherit **FST\_EXE** from the **\_MAIN** block.

The following macros are available for a PACSystems folder:

#### Transition and Alarm Macros

#### Macros for accessing the %I, %Q, %M, %T, %G, %S, and %SA - %SC transition bits

---

**Note:** A transition bit is set high (1) if consecutive writes to a reference bit results in the bit transitioning from a 0 to 1 or 1 to 0. The bit is cleared (0) if consecutive writes to a reference bit result in the bit staying at the same state (0 to 0, 1 to 1, for example).

---

```

BIT_TST_I_TRANS(x)
BIT_TST_Q_TRANS(x)
BIT_TST_M_TRANS(x)
BIT_TST_T_TRANS(x)
BIT_TST_G_TRANS(x)
BIT_TST_S_TRANS(x)
BIT_TST_SA_TRANS(x)
BIT_TST_SB_TRANS(x)
BIT_TST_SC_TRANS(x)

```

## Macros for accessing the %I, %Q, %M, %T, %G, %S, and %SA - %SC transition bits as bytes

IB\_TRANS(x)  
QB\_TRANS(x)  
MB\_TRANS(x)  
TB\_TRANS(x)  
GB\_TRANS(x)  
SB\_TRANS(x)  
SAB\_TRANS(x)  
SBB\_TRANS(x)  
SCB\_TRANS(x)

## Macros for accessing the %I, %Q, %AI, %AQ Diagnostic memory

### Definitions used with macros that access Analog Input DIAGNOSTIC memory(s)

HI\_ALARM\_MSK 0x02  
LO\_ALARM\_MSK 0x01  
AI\_OVERRANGE\_MSK 0x08  
AI\_UNDERRANGE\_MSK 0x04

### Definitions used with macros that access Analog Output DIAGNOSTIC memory(s)

AQ\_OVERRANGE\_MSK 0x40  
AQ\_UNDERRANGE\_MSK 0x20

### Diagnostic memory macros

---

**Note:** Discrete diagnostic memory is organized so that there is one fault bit per discrete memory location. Analog diagnostic memory is organized so that there is one byte of memory for each analog input or output channel (for example there is one diagnostic byte associated with the analog input %AQ1). For analog diagnostic memory, use the mask definitions above to determine the type of analog fault for a particular analog input or output channel.

---

BIT\_TST\_I\_DIAG(x)  
BIT\_TST\_Q\_DIAG(x)  
IB\_DIAG(x)  
QB\_DIAG(x)  
AIB\_DIAG(x)  
AQB\_DIAG(x)  
AI\_HIALRM(x)  
AI\_LOALRM(x)

---

**Note:** AIB\_FAULT and AQB\_FAULT are non-zero for conditions that set a fault contact or generate a fault entry in the I/O fault table such as Overrange, Underrange.

---

AIB\_FAULT(x)  
AQB\_FAULT(x)  
AI\_OVERRANGE(x)

AI\_UNDERRANGE(x)  
AQ\_OVERRANGE(x)  
AQ\_UNDERRANGE(x)

## Macros for accessing RACK/SLOT/BLOCK fault information

See descriptions of the corresponding functions in Section 3.5.10, Reference Memory Functions.

Macro Name		page
RACKX(r)	rackX(r)	107
SLOTX(r,s)	slotX(r,s)	108
BLOCKX(r,s,b,sba)	blockX(r,s,b,sba)	108
RSMB(x)	rsmb(x)	109

## 3.5 Standard Library Routines

Appendix A contains a complete list of the standard C library routines supported by C blocks. The routines implement ANSI C functionality unless otherwise noted.

The printf function is not supported. You should use the message mode functions described later in this section to access the PLC serial port.

### 3.5.1 PACSystems Functions

Additional functions are provided by the C Toolkit in support of the PACSystems CPU's operations. These functions are defined in the header file included by PACRxPLC.h. These header files are:

Header File	Functions	Page
ctkPlcBus.h	Bus Read/Write Functions	39
ctkPlcErrno.h	Errno Functions	112
ctkPlcFault.h	Fault Table Service Request Functions	84
ctkPlcFunc.h	General PLC Functions	33
	Miscellaneous General Functions	93
	Service Request Functions	53
ctkPlcUtil.h	Utility Function	111
ctkRefMem.h	Reference Memory Functions	95
ctkVariables.h	PLC Variable Access	113

These files are located in the following subdirectory:

PACSystemsCtoolkit\Targets\CommonFiles\IncCommon

Descriptions of the functions are provided in the sections that follow.

## 3.5.2 General PLC Functions

The following functions make PLC features available to C applications. These functions are described in `ctkPlcFunc.h`.

### PLCC\_read\_elapsed\_clock

`T_INT32 PLCC_read_elapsed_clock(struct elapsed_clock_rec *pElapsedClockRec);`

```
struct elapsed_clock_rec {  
    T_DWORD seconds  
    T_WORD hundred μsecs;  
};
```

#### Description

This function returns the current time from the PLC in memory pointed to by `pElapsedClockRec`, which is the time since the PLC powered up.

#### InParam pElapsedClockRec

Pointer to structure containing the value of the PLC's elapsed clock

#### RetVal

The return value is 0 if successful, -1 if unsuccessful.

### PLCC\_read\_nano\_elapsed\_clock

`T_INT32 PLCC_read_nano_elapsed_clock(struct nano_elapsed_clock_rec  
 *pNanoElapsedClockRec);`

```
struct nano_elapsed_clock_rec {  
    T_DWORD  seconds  
    T_DWORD  nanoseconds;  
};
```

#### Description

This function returns the current time, in nanosecond units, from the PLC in memory pointed to by `pNanoElapsedClockRec`, which is the time since the PLC powered up.

#### InParam pNanoElapsedClockRec

Pointer to structure containing the value of the PLC's elapsed clock in nanosecond units.

#### RetVal

The return value is 0 if successful, -1 if unsuccessful.

## PLCC\_chars\_in\_printf\_q

**Obsolete:** Use Proc PLCC\_CharsInMessageWriteQ function.

T\_INT32 PLCC\_chars\_in\_printf\_q(void);

This function returns GEF\_NOT\_SUPPORTED.

## PLCC\_MessageWrite

T\_INT32 PLCC\_MessageWrite(T\_INT32 port, char \*buffer, T\_INT32 numBytes);

#define PORT1 0

#define PORT2 1

### Description

Writes to a serial port on the PLC.

### InParam port

Indicates which PLC serial port to write (i.e. PORT1, PORT2).

### InParam buffer

Pointer to the buffer of data to write to the serial port.

### InParam numBytes

Number of bytes to write (up to MESSAGE\_BUFFER\_SIZE).

### ReturnVal

If successful, returns the number of bytes written. This may be less than the number of bytes requested if the write queue fills.

Returns -1 for a bad parameter or if message mode is not configured for the specified port.

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.



## Proc PLCC\_MessageRead

T\_INT32 PLCC\_MessageRead(T\_INT32 port, char \*buffer, T\_INT32 numBytes);

### Description

Reads from the serial port input queue on the PLC.

### InParam port

Indicates which PLC serial port to read (i.e. PORT1, PORT2).

### InParam buffer

Pointer to the buffer to place the data read from the input queue.

### InParam numBytes

Number of bytes to read (up to MESSAGE\_BUFFER\_SIZE).

### ReturnVal

If successful, returns the number of bytes read. This may be less than the number of bytes requested if it is larger than the number of bytes in the read queue.

Returns -1 for a bad parameter or if message mode is not configured for the specified port.

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## Proc PLCC\_CharsInMessageWriteQ

T\_INT32 PLCC\_CharsInMessageWriteQ(T\_INT32 port);

### Description

Returns the number of bytes in the write queue.

### InParam port

Indicates which PLC serial port to query (i.e. PORT1, PORT2).

### ReturnVal

If successful, returns the number of bytes in the queue.

Returns -1 for a bad parameter or if message mode is not configured for the specified port.

### Errno

If there is an error, Errno is set by this function to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## Proc PLCC\_CharsInMessageReadQ

T\_INT32 PLCC\_CharsInMessageReadQ(T\_INT32 port);

### Description

Returns the number of bytes in the read queue.

### InParam port

Indicates which PLC serial port to query (i.e. PORT1, PORT2).

### ReturnVal

If successful, returns the number of bytes in the queue.

Returns -1 for a bad parameter or if message mode is not configured for the specified port.

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## PLCC\_gen\_alarm

T\_INT32 PLCC\_gen\_alarm(T\_WORD error\_code, char \*fault\_string);

### Description

This function puts the fault described by **error\_code** and **fault\_string** into the controller fault table.

### InParam error\_code

Indicates the user specified error that is to be logged. The error code must be less than 2048.

### InParam fault\_string

Pointer to a character string describing the fault. String must be NULL terminated and less than 24 characters.

### ReturnVal

This function will return 0 if successful and -1 if unsuccessful.

## PLCC\_get\_plc\_version

T\_INT32 PLCC\_get\_plc\_version(struct PLC\_ver\_info\_rec \*PLC\_ver\_info);

\*\*\* ALL DATA RETURNED FROM THE PLC (in the structure

PLC\_ver\_info) NEEDS TO BE LOOKED AT IN HEXADECIMAL

for proper interpretation

```
struct PLC_ver_info_rec {
    T_WORD      family; /* Host PLC product line */
    T_WORD      model;   /* Specific Model of PLC */
    T_BYTE sw_ver; /* Major Version of PLC firmware */
    T_BYTE sw_rev; /* Minor Revision of PLC firmware */
};

/* Family value */
#define FAMILY_PACSYSTEMS 0x2002
/* Model numbers */
#define CPE_010 0x02 /* PACSystems RX7i 300MHz PLC CPU */
#define CPE_020 0x04 /* PACSystems RX7i 700MHz PLC CPU */
#define CRE_020 0x05 /* PACSystems RX7i 700MHz Redundant PLC CPU */
#define CPE_030 0x06 /* PACSystems RX7i VME 700MHz (Pentium M) PLC CPU */
#define CPE_040 0x08 /* PACSystems RX7i VME 1.8GHz (Pentium M) PLC CPU */
#define CPU_310 0x0A /* PACSystems Rx3i PCI 300MHz PLC CPU */
#define CPE_310 0x0A /* PACSystems Rx3i PCI 1.1GHz (Atom) PLC CPU (10 MB) */
#define NIU_001 0x0C /* PACSystems Rx3i PCI 300MHz NIU */
#define CMU_310 0x0E /* PACSystems Rx3i PCI 300MHz MaxOn CPU */
#define CPE_330 0x10 /* PACSystems Rx3i CPE330 */
#define CPE_305 0x12 /* PACSystems Rx3i PCI 1.1GHz (Atom) PLC CPU (5 MB) */
#define CPE_400 0x26 /* PACSystems Rx3i CPE400 */
#define EPSCPE100 0x27 /* PACSystems EPS CPE100 CPU */
#define EPSCPE115 0x28 /* PACSystems EPS CPE115 CPU */
#define CPE_302 0x2A /* PACSystems Rx3i PCI 1.1GHz (Atom) PLC CPU (2 MB) */
#define CPL_410 0x2C /* PACSystems Rx3i IICS CPU2 with Open Linux */
#define CPE_302 0xXX /* PACSystems Rx3i CPE302 */
```

## Description

This function returns the PLC family, model, firmware version, and firmware revision.

## InParam PLC\_ver\_info

Pointer to the structure of type PLC\_ver\_info. The PLC will return information concerning its firmware version in each of the fields in this structure.

## ReturnVal

The function will return 0 if successful and -1 if unsuccessful.

### 3.5.3 Bus Read/Write Functions

The following functions based on the BUS functions available in ladder logic are defined in ctkPlcBus.h. These functions are currently unsupported in the Rx3i and will return a not-supported return value (-1). When reading the memory pointed to by pStatus the following values are possible variables returned by these functions:

Variable	Numeric Value
NOT_SUPPORTED	-1
OPERATION_SUCCESSFUL	0
BUS_ERROR	1
MOD_DOES_NOT_EXIST	2
INVALID_MOD	3
START_ADDR_RANGE_ERR	4
END_ADDR_RANGE_ERR	5
EVEN_ADDR_ODD_CONFIG_ERR	6
ODD_ADDR_EVEN_CONFIG_ERR	7
WINDOW_NOT_ENABLED	8
INVALID_ACCESS_WIDTH	9
INVALID_PARAM	10

#### Note:

1. The hardware configuration must be set up for the largest access for these functions to complete with a successful status. For example, the module memory region Interface Type must use Dword Access if any of the Dword functions are used. However, if only Word or Byte functions are used, the Interface type can be Word Access. Similarly, if only byte functions are used, the Interface type can be Byte Access. In addition, Word Access functions must use only even addresses and Dword Access functions must be Dword aligned (0, 4, 8, etc.).
2. The subSlot value for most modules will be 0.

## Proc PLCC\_BUS\_read\_byte

T\_INT32 PLCC\_BUS\_read\_byte(T\_WORD rack, T\_WORD slot, T\_WORD subSlot,  
T\_WORD region, T\_WORD \*pStatus,  
T\_BYTE \*pBuffer, T\_DWORD address);

### Description

Read a byte from a device on the bus.

### InParam rack

The rack number containing the module to access.

### InParam slot

The slot number containing the module to access.

### InParam subSlot

The sub-slot number of the module to access.

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### OutParam pBuffer

Pointer to the byte read in from a device on the bus.

### InParam address

Address of the byte to be read.

### ReturnVal

1 if successful

0 if unsuccessful

-1 if not supported

## Proc PLCC\_BUS\_read\_word

```
T_INT32 PLCC_BUS_read_word(T_WORD rack, T_WORD slot, T_WORD subSlot,  
    T_WORD region, T_WORD *pStatus,  
    T_WORD *pBuffer, T_DWORD address);
```

### Description

Read a word from a device on the bus.

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### OutParam pBuffer

Pointer to the word read in from a device on the bus.

### InParam address

Address of the word to be read.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_read\_dword

```
T_INT32 PLCC_BUS_read_dword(T_WORD rack, T_WORD slot, T_WORD subSlot,  
    T_WORD region, T_WORD *pStatus,  
    T_DWORD *pBuffer, T_DWORD address);
```

### Description

Read a Dword from a device on the bus.

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### OutParam pBuffer

Pointer to the Dword read in from a device on the bus.

### InParam address

Address of the Dword to be read.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported



## Proc PLCC\_BUS\_read\_block

```
T_INT32 PLCC_BUS_read_block(T_WORD rack, T_WORD slot, T_WORD subSlot,  
                             T_WORD region, T_WORD *pStatus,  
                             void *pBuffer, T_WORD length,  
                             T_DWORD address);
```

### Description

Read a block from a device on the bus.

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### OutParam pBuffer

Pointer to the data area to put the data.

### InParam length

Size of the data area in bytes.

### InParam address

Start Address of the data area to be read.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_write\_byte

T\_INT32 PLCC\_BUS\_write\_byte(T\_WORD rack, T\_WORD slot, T\_WORD subSlot,  
T\_WORD region, T\_WORD \*pStatus,  
T\_BYTE value, T\_DWORD address);

### Description

Write a byte to a device on the bus.

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### InParam value

Byte value to be written to a device on the bus.

### InParam address

Address of the byte to be written.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_write\_word

```
T_INT32 PLCC_BUS_write_word(T_WORD rack, T_WORD slot, T_WORD subSlot,  
                             T_WORD region, T_WORD *pStatus,  
                             T_WORD value, T_DWORD address);
```

### Description

Write a word to a device on the bus.

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### InParam value

Word value to be written to a device on the bus.

### InParam address

Address of the word to be **written**.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_write\_dword

T\_INT32 PLCC\_BUS\_write\_dword(T\_WORD rack, T\_WORD slot, T\_WORD subSlot,  
T\_WORD region, T\_WORD \*pStatus,  
T\_DWORD value, T\_DWORD address);

### Description

Write a Dword to a device on the bus.

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### InParam value

Dword value to be written to a device on the bus.

### InParam address

Address of the Dword to be written.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_write\_block

```
T_INT32 PLCC_BUS_write_block(T_WORD rack, T_WORD slot, T_WORD subSlot,  
                             T_WORD region, T_WORD *pStatus,  
                             void *pBuffer, T_WORD length,  
                             T_DWORD address);
```

### Description

Write a block of data to a device on the bus

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### InParam pBuffer

Pointer to the data to be written to a device on the bus.

### InParam length

Length of the data to be written to a device on the bus in bytes.

### InParam address

Address of the data to be written.

### RetVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## 3.5.4 BUS Semaphore Functions

The following functions are designed to enable semaphore handling on the bus. These functions cannot be interrupted.

BUS\_RMW (read, modify, write)

---

**Note:** The following definitions are used to define whether the mask parameter uses an OR AND operation on the data: BUS\_OR, BUS\_AND.

---

### PLCC\_BUS\_RMW\_byte

```
T_INT32 PLCC_BUS_RMW_byte(T_WORD rack, T_WORD slot, T_WORD subSlot,  
    T_WORD region, T_WORD *pStatus,  
    T_BYTE *pOriginalValue, T_WORD op_type,  
    T_DWORD mask, T_DWORD address);
```

#### Description

Read Modify Write a byte to a device on the bus.

#### InParam rack

The rack number containing the module to access

#### InParam slot

The slot number containing the module to access

#### InParam subSlot

The sub-slot number of the module to access

#### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

#### OutParam pStatus

Pointer to status variable (see common definition above)

#### OutParam pOriginalValue

Pointer to the value before the read-modify-write operation

#### InParam op\_type

Specifies whether the mask is ANDed or ORed with the data. BUS\_OR or BUS\_AND

#### InParam mask

Data mask.

## InParam address

Address of the data to be written.

## ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_RMW\_word

```
T_INT32 PLCC_BUS_RMW_word(T_WORD rack, T_WORD slot, T_WORD subSlot,  
                           T_WORD region, T_WORD *pStatus,  
                           T_WORD *pOriginalValue, T_WORD op_type,  
                           T_DWORD mask, T_DWORD address);
```

## Description

Read Modify Write a word to a device on the bus

## InParam rack

The rack number containing the module to access

## InParam slot

The slot number containing the module to access

## InParam subSlot

The sub-slot number of the module to access

## InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

## OutParam pStatus

Pointer to status variable (see common definition above).

## OutParam pOriginalValue

Pointer to the value before the read-modify-write operation.

## InParam op\_type

Specifies whether the mask is ANDed or ORed with the data. BUS\_OR or BUS\_AND

## InParam mask

Data mask.

## InParam address

Address of the data to be written.

## ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_RMW\_dword

```
T_INT32 PLCC_BUS_RMW_dword(T_WORD rack, T_WORD slot, T_WORD subSlot,  
                             T_WORD region, T_WORD *pStatus,  
                             T_DWORD *pOriginalValue, T_WORD op_type,  
                             T_DWORD mask, T_DWORD address);
```

## Description

Read Modify Write a dword to a device on the bus

## InParam rack

The rack number containing the module to access

## InParam slot

The slot number containing the module to access

## InParam subSlot

The sub-slot number of the module to access

## InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

## OutParam pStatus

Pointer to status variable (see common definition above)

## OutParam pOriginalValue

Pointer to the value before the read-modify-write operation

## InParam op\_type

Specifies whether the mask is ANDed or ORed with the data. BUS\_OR or BUS\_AND

## InParam mask

Data mask.



## InParam address

Address of the data to be written.

## ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_TST\_byte

```
T_INT32 PLCC_BUS_TST_byte(T_WORD rack, T_WORD slot, T_WORD subSlot,  
                           T_WORD region, T_WORD *pStatus,  
                           T_BYTE *pSemaphoreOutput, T_DWORD address);
```

## Description

This function reads a byte sized semaphore from the bus address and tests the least significant bit. The semaphore output will be 0 if the semaphore is not obtained, 1 if it is obtained. You must release this semaphore when it is no longer needed. To release a semaphore, write 0 to the semaphore.

## InParam rack

The rack number containing the module to access

## InParam slot

The slot number containing the module to access

## InParam subSlot

The sub-slot number of the module to access

## InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

## OutParam pStatus

Pointer to status variable (see common definition above)

## OutParam semaphore\_output

Results of locking semaphore

0 = not obtained  
1 = obtained

## InParam address

Address of the data to be written.

## ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Proc PLCC\_BUS\_TST\_word

T\_INT32 PLCC\_BUS\_TST\_word(T\_WORD rack, T\_WORD slot, T\_WORD subSlot,  
T\_WORD region, T\_WORD \*pStatus,  
T\_WORD \*pSemaphoreOutput, T\_DWORD address);

### Description

This function reads a word-sized semaphore from the bus address and tests the least significant bit. The semaphore output will be 0 if the semaphore is not obtained, 1 if it is obtained. You must free this semaphore when it is no longer needed. To release a semaphore, write 0 to the semaphore. The address must be word-aligned.

### InParam rack

The rack number containing the module to access

### InParam slot

The slot number containing the module to access

### InParam subSlot

The sub-slot number of the module to access

### InParam region

The region number describing the location of the BUS memory. This is set up in hardware configuration for the module.

### OutParam pStatus

Pointer to status variable (see common definition above)

### OutParam semaphore\_output

Results of locking semaphore

0 = not obtained  
1 = obtained

### InParam address

Address of the data to be written.

## RetVal

1 if successful  
0 if unsuccessful  
-1 if not supported

### 3.5.5 Service Request Functions

The following functions are patterned after the service request (SVC\_REQ) function in ladder logic and defined in ctkPlcFunc.h.

#### PLCC\_const\_sweep\_timer

```
T_INT32 PLCC_const_sweep_timer(struct const_sweep_timer_rec
                                *pConstSweepTimerRec);

/* input structure */
struct const_sweep_input_rec {
    T_WORD    action;
    T_WORD    timer_value;
};

/* structure with return value */
struct const_sweep_output_rec {
    T_WORD    sweep mode;
    T_WORD    current_time_value;
};

struct const_sweep_timer_rec {
    union {
        struct  const_sweep_input_rec input;
        struct  const_sweep_output_rec output;
    };const_sweep; /*Note: union name required with PACSystems */
};

/* sweep mode values - these determine which action is to be taken */
#define DISABLE_CONSTANT_SWEEP_MODE    0
#define ENABLE_CONSTANT_SWEEP_MODE     1
#define CHANGE_TIMER_VALUE             2
#define READ_TIMER_VALUE_AND_STATE     3

/* sweep mode return values */
#define CONSTANT_SWEEP_ENABLED         1
#define CONSTANT_SWEEP_DISABLED        0
```

## Description

This function is the C interface to service request #1 (Change/Read Constant Sweep Timer).

This function can be used to

- Disable constant sweep time mode
- Enable constant sweep time mode and use the old timer value
- Enable constant sweep time mode and use a new timer value
- Set a new timer value only
- Read constant sweep mode state timer and value

Setting sweep\_mode to DISABLE\_CONSTANT\_SWEEP\_MODE disables the constant sweep timer. Setting sweep\_mode to ENABLE\_CONSTANT\_SWEEP\_MODE enables the constant with the value in sweep\_timer or keep the current value if the sweep\_timer is 0. Setting the sweep\_mode to CHANGE\_TIMER\_VALUE changes the constant sweep timer to the value in timer\_value. Setting sweep\_mode to READ\_TIMER\_VALUE\_AND\_STATE sets sweep\_enabled to 1 if the constant sweep timer is enabled and sets the current constant sweep timer value to the current\_value.

## In/OutParam pConstSweepTimerRec

Pointer to structure containing constant sweep timer record.

## RetVal

This function returns 1 if successful and 0 if unsuccessful, and -1 if not supported.

## PLCC\_read\_window\_values

```
T_INT32 PLCC_read_window_values(struct read_window_values_rec
                                *pReadWindowValuesRec);
```

```
/* window modes */

#define LIMITED_MODE      0
#define CONSTANT_MODE    1
#define RUN_TO_COMPLETION_MODE 2

/* structure with return values */
struct read_window_values_rec{
    T_BYTE controller_win_time;
    T_BYTE controller_win_mode; /* LIMITED_MODE, CONSTANT_MODE,
                                RUN_TO_COMPLETION_MODE */
    T_BYTE backplane_comm_win_time;
    T_BYTE backplane_comm_win_mode; /* LIMITED_MODE, CONSTANT_MODE,
                                    RUN_TO_COMPLETION_MODE */
    T_BYTE background_win_time;
```

```
T_BYTE background_win_mode; /* LIMITED_MODE, CONSTANT_MODE, */
                          /* RUN_TO_COMPLETION_MODE */
};
```

## Description

This function is the C interface to service request #2 (Read Window Values). This function will return the mode and time for the controller communications window, the backplane communications window, and the background task window in the structure.

---

**Note:** *The Series 90-70 referred to the Controller Communications window as the Programmer Communications window. Also, the 90-70 referred to the Backplane Communications window as the System Communications Window.*

---

The possible values for the mode fields are LIMITED\_MODE, CONSTANT\_MODE, and RUN\_TO\_COMPLETION\_MODE. The time fields contain the time values in milliseconds.

## OutParam pStatus

Pointer to structure containing record of the read window values.

## RetVal

This function will return 1 if successful and 0 if unsuccessful.

## PLCC\_change\_controller\_comm\_window

T\_INT32 PLCC\_change\_controller\_comm\_window

```
(struct change_controller_comm_window_rec
 *pChangeControllerCommWindowRec);
```

```
struct change_controller_comm_window_rec{
    T_BYTE time;
    T_BYTE mode; /* LIMITED_MODE, CONSTANT_MODE, */
                /* RUN_TO_COMPLETION_MODE */
};
```

## Description

This function is the C interface to service request #3 (Change Controller Communications Window State and Values).

---

**Note:** *The Series 90-70 documentation refers to the Controller Communications window as the Programmer Communications window.*

---

This function will change the Controller communications window state and timer to the values specified in the structure. The mode will be changed to one of the three states LIMITED\_MODE, CONSTANT\_MODE, or RUN\_TO\_COMPLETION\_MODE depending on the value in the mode field.

## InParam pChangeControllerCommWindowRec

Pointer to structure containing change controller window record. The time value should be from 1 to 255 milliseconds.

## RetVal

This function will return 1 if successful and 0 if unsuccessful.

## PLCC\_change\_backplane\_comm\_window

T\_INT32 PLCC\_change\_backplane\_comm\_window(struct  
change\_system\_comm\_window\_rec

\*pChangeBackplaneCommWindowRec);

struct change\_system\_comm\_window\_rec {

    T\_BYTE time;

    T\_BYTE mode;

};

/\* window modes \*/

#define LIMITED\_MODE 0

#define CONSTANT\_MODE 1

#define RUN\_TO\_COMPLETION\_MODE 2

## Description

This function is the C interface to service request #4 (Change Backplane Communications Window State and Values).

---

**Note:** *The Series 90-70 documentation refers to the Controller Communications window as the Programmer Communications window.*

---

This function will change the Backplane Communications Window state and timer to the values specified in the structure. The mode will be changed to one of the three states LIMITED\_MODE, CONSTANT\_MODE, or RUN\_TO\_COMPLETION\_MODE depending on the value in the mode field.

## InParam pChangeBackplaneCommWindowRec

Pointer to structure containing backplane communications record. The time value should be from 1 to 255 milliseconds.

## RetVal

This function will return 1 if successful and 0 if unsuccessful.

## PLCC\_change\_background\_window

```
T_INT32 PLCC_change_background_window(struct change_background_window_rec  
                                     *pChangeBackgroundWindowRec);
```

```
struct change_background_window_rec {  
    T_BYTE time;  
    T_BYTE mode;  
};  
  
/* window modes */  
#define LIMITED_MODE                0  
#define CONSTANT_MODE              1  
#define RUN_TO_COMPLETION_MODE     2
```

### Description

This function is the C interface to service request #5 (Change\_Background Window State and Values). This function will change the background window state and timer to the values specified in the structure. The mode will be changed to one of the three states LIMITED\_MODE, CONSTANT\_MODE, or RUN\_TO\_COMPLETION\_MODE depending on the value in the mode field.

### InParam pChangeBackgroundWindowRec

Pointer to structure containing background window record. The time value should be from 1 to 255 milliseconds.

### RetVal

This function will return 1 if successful and 0 if unsuccessful.

## PLCC\_number\_of\_words\_in\_chksm

```
T_INT32 PLCC_number_of_words_in_chksm(struct number_of_words_in_chksm_rec
                                     *pNumberOfWordsInChksmRec);
```

```
struct number_word_of_words_in_chksm_rec {
    T_WORD    read_set;
    T_WORD    word_count;
};

#define READ_CHECKSUM_WORDS    0
#define SET_CHECKSUM_WORDS    1
```

### Description

This function is the C interface to service request #6 (Change/Read Checksum Task State and Number of Words to Checksum). This function will either read the current checksum word count or set a new checksum word count depending on the value in read\_set. If read\_set is READ\_CHECKSUM, then the function will read the current word count and return it in word\_count. If the read\_set is SET\_CHECKSUM, then the function will set the current word count to word\_count rounded to the nearest multiple of 8. To disable the checksums set the word\_count to 0. The function will fail if the read\_write field is set to a value other than 0 or 1.

### InParam pNumberOfWordsInChksmRec

Pointer to structure containing number of words in checksum record.

### ReturnVal

This function will return 1 if successful and 0 if unsuccessful.



## PLCC\_tod\_clock

T\_INT32 PLCC\_tod\_clock(struct tod\_clock\_rec \*pTodClockRec);

### Data Formats

This function supports the following data formats:

```
#define NUMERIC_DATA_FORMAT      0
#define BCD_FORMAT                1
#define UNPACKED_BCD_FORMAT      2
#define PACKED_ASCII_FORMAT      3
#define POSIX_FORMAT              4

#define NUMERIC_DATA_FORMAT_4_DIG_YR    0x80
#define BCD_FORMAT_4_DIG_YR            0x81
#define UNPACKED_BCD_FORMAT_4_DIG_YR    0x82
#define PACKED_ASCII_FORMAT_4_DIG_YR    0x83
```

### Day of the Week Definitions:

```
#define SUNDAY          1
#define MONDAY          2
#define TUESDAY         3
#define WEDNESDAY       4
#define THURSDAY        5
#define FRIDAY          6
#define SATURDAY        7
```

### NUMERIC\_DATA\_FORMAT

Decimal values for fields. For example, '94 for the year would be 94 decimal in the year field.

```
struct num_tod_rec{
    T_WORD year;
    T_WORD month;
    T_WORD day_of_month;
    T_WORD hours; T_WORD minutes;
    T_WORD seconds;
    T_WORD day_of_week;
};
```

### BCD\_FORMAT

Hexadecimal values for the fields. For example, '94 for the year would be 0x94.

```
struct BCD_tod_rec{
    T_BYTE year;
    T_BYTE month;
    T_BYTE day_of_month;
    T_BYTE hours;
    T_BYTE minutes;
    T_BYTE seconds;
    T_BYTE day_of_week;
    T_BYTE null;
};
```

```
struct BCD_tod_4_rec{
    T_BYTE year_lo;
    T_BYTE year_hi;
    T_BYTE month;
    T_BYTE day_of_month;
    T_BYTE hours;
    T_BYTE minutes;
    T_BYTE seconds;
    T_BYTE day_of_week;
};
```

### UNPACKED\_BCD\_FORMAT

Two byte fields make up the word category. For example, '94 for the year is 9 in yearhi and 4 in yearlo.

```
struct unpacked_BCD_rec{
    T_BYTE yearlo;
    T_BYTE yearhi;
    T_BYTE monthlo;
    T_BYTE monthhi;
    T_BYTE day_of_month_lo;
    T_BYTE day_of_month_hi;
};
```

```

    T_BYTE hourslo;
    T_BYTE hourshi;
    T_BYTE minslo;
    T_BYTE minshi;
    T_BYTE secslo;
    T_BYTE secshi;
    T_WORD day_of_week;
};

struct unpacked_bcd_tod_4_rec{
    T_WORD huns_year;
    T_WORD tens_year;
    T_WORD month;
    T_WORD day_of_month;
    T_WORD hours;
    T_WORD minutes;
    T_WORD seconds;
    T_WORD day_of_week;
};

```

#### PACKED\_ASCII\_FORMAT

Two ASCII character fields make up the word category. For example, 94 for the year is '9' in yearhi and '4' in yearlo.

```

struct ASCII_tod_rec{
    T_BYTE yearhi;
    T_BYTE yearlo;
    T_BYTE space1;
    T_BYTE monthhi;
    T_BYTE monthlo;
    T_BYTE space2;
    T_BYTE day_of_month_hi;
    T_BYTE day_of_month_lo;
    T_BYTE space3;
    T_BYTE hourshi;
    T_BYTE hourslo;
}

```

```
T_BYTE colon1;
T_BYTE minshi;
T_BYTE minslo;
T_BYTE colon2;
T_BYTE secshi;
T_BYTE secslo;
T_BYTE space4;
T_BYTE day_of_week_hi;
T_BYTE day_of_week_lo;
};
struct ascii_tod_4_rec{
    T_BYTE hun_year_hi;
    T_BYTE hun_year_lo;
    T_BYTE year_hi;
    T_BYTE year_lo;
    T_BYTE space1;
    T_BYTE month_hi;
    T_BYTE month_lo;
    T_BYTE space2;
    T_BYTE day_of_month_hi;
    T_BYTE day_of_month_lo;
    T_BYTE space3;
    T_BYTE hours_hi;
    T_BYTE hours_lo;
    T_BYTE colon1;
    T_BYTE minutes_hi;
    T_BYTE minutes_lo;
    T_BYTE colon2;
    T_BYTE seconds_hi;
    T_BYTE seconds_lo;
    T_BYTE space4;
    T_BYTE day_of_week_hi;
    T_BYTE day_of_week_lo;
```

```
};  
  
/* Definitions to be used with "read_write" field */  
READ_CLOCK      0  
WRITE_CLOCK     1  
  
struct tod_clock_rec{  
    T_WORD read_write; /* READ_CLOCK or WRITE_CLOCK */  
    T_WORD format;     /* NUMERIC_DATA_FORMAT, BCD_FORMAT,  
                        UNPACKED_BCD_FORMAT, PACKED_ASCII_FORMAT etc.  
                        (see above for additional formats)  
                        Note: All formats may not be supported by a  
                        particular PLC target */  
    union {  
        struct num_tod_rec num_tod;  
        struct BCD_tod_rec BCD_tod;  
        struct BCD_tod_4_rec BCD_tod_4;  
        struct unpacked_BCD_rec unpacked_BCD_tod;  
        struct unpacked_bcd_tod_4_rec unpacked_BCD_tod_4;  
        struct ASCII_tod_rec ASCII_tod;  
        struct ascii_tod_4_rec ASCII_tod_4;  
        struct timespec POSIX_tod; /* timespec is defined in sys/types.h */  
    } record; /* Note: 90-70 C Toolkit did not name this union */  
};
```

## Description

This function is the C interface to service request #7 (Change/Read Time-of-Day Clock State and Values). If `read_write` is equal to `READ_CLOCK`, then the function will read the Time-of-Day Clock into the structure passed. If `read_write` is equal to `WRITE_CLOCK`, then the function will write the values in the structure to the `time_of_day_clock`. The format will be based on the `format` field in the structure (`NUMERIC_DATA_FORMAT`, `BCD_FORMAT`, `UNPACKED_BCD_FORMAT`, and `PACKED_ASCII_FORMAT`). The function will fail in the following instances:

- If `read_write` is some number other than 0 or 1
- If `format` is some number other than 0 – 3
- If data for a write does not match format

For all the formats, the hours are 24-hour and the days of the week are defined as macros in `ctkFuncPlc.h`. The packed BCD format needs the null field to be 0, as shown in the following example:

Example:

```
#include "PACRxPLC.h"

int GeFMain()
{
    struct tod_clock_rec data;

    data.read_write = 1;
    data.format = BCD_FORMAT;

    /* set the time and date to 1:13:08pm Tuesday August 9, 1994 */
    data.record.BCD_tod.year = 0x94;
    data.record.BCD_tod.month = 8;
    data.record.BCD_tod.day_of_month = 9;
    data.record.BCD_tod.hours = 0x13;
    data.record.BCD_tod.minutes = 0x13;
    data.record.BCD_tod.seconds = 8;
    data.record.BCD_tod.day_of_week = TUESDAY;
    data.record.BCD_tod.null = 0;
    PLCC_tod_clock(&data)
}
```

The unpacked format should have a digit in every byte (including the day of the week) as shown in the following example:

```
Example:
#include "PACRxPLC.h"

int GeFMain()
{
    struct tod_clock_rec data;

    data.read_write = 1;
    data.format = UNPACKED_BCD_FORMAT;

    /* set the time and date to 1:13:08pm Tuesday August 9, 1994 */
    data.record.unpacked_BCD_tod.yearhi = 9;
    data.record.unpacked_BCD_tod.yearlo = 4;
    data.record.unpacked_BCD_tod.monthhi = 0;
    data.record.unpacked_BCD_tod.monthlo = 8;
    data.record.unpacked_BCD_tod.day_of_month_hi = 0;
    data.record.unpacked_BCD_tod.day_of_month_lo = 9;
    data.record.unpacked_BCD_tod.hourshi = 1;
    data.record.unpacked_BCD_tod.hourslo = 3;
    data.record.unpacked_BCD_tod.minshi = 1;
    data.record.unpacked_BCD_tod.minslo = 3;
    data.record.unpacked_BCD_tod.secshi = 0;
    data.record.unpacked_BCD_tod.secslo = 8;
    data.record.unpacked_BCD_tod.day_of_week = TUESDAY;
    PLCC_tod_clock (& data)
}
```

The packed ASCII format should have an ASCII character in every byte as shown in the following example:

Example:

```
#include "PACRx PLC.h"

int GeFMain()
{
    struct tod_clock_rec data;

    data.read_write = 1;
    data.format = PACKED_ASCII_FORMAT;

    /* set the time and date to 1:13:08pm Tuesday August 9, 1994 */
    data.record.ASCII_tod.yearhi = '9';
    data.record.ASCII_tod.yearlo = '4';
    data.record.ASCII_tod.space1 = ' ';
    data.record.ASCII_tod.monthhi = '0';
    data.record.ASCII_tod.monthlo = '8';
    data.record.ASCII_tod.space2 = ' ';
    data.record.ASCII_tod.day_of_month_hi = '0';
    data.record.ASCII_tod.day_of_month_lo = '9';
    data.record.ASCII_tod.space3 = ' ';
    data.record.ASCII_tod.hourshi = '1';
    data.record.ASCII_tod.hourslo = '3';
    data.record.ASCII_tod.colon1 = ':';
    data.record.ASCII_tod.minshi = '1';
    data.record.ASCII_tod.minslo = '3';
    data.record.ASCII_tod.colon2 = ':';
    data.record.ASCII_tod.secshi = '0';
    data.record.ASCII_tod.secslo = '8';

    /* place 0 ASCII (30 hex) in the high byte for the number */
    data.record.ASCII_tod.day_of_weekhi = '0';

    /* place TUESDAY(3) plus 30 hex into the lo          */
    /* byte to make the number an ASCII character      */
    data.record.ASCII_tod.day_of_weeklo = TUESDAY+0x30;
    PLCC_tod_clock_rec (& data)
}
```



## In/OutParam pTodClockRec

Pointer to structure containing time of day clock record.

## RetVal

This function returns 1 if successful, 0 if unsuccessful or -1 if not supported.

## PLCC\_reset\_watchdog\_timer

T\_INT32 PLCC\_reset\_watchdog\_timer(void);

## Description

This function is the C interface to service request #8 (Reset Watchdog Timer). This function will reset the watchdog timer during the sweep. When the watchdog timer expires, the PLC shuts down without warning. This function allows the timer to be refreshed during a time-consuming task.

## RetVal

This function will return 1 if successful and 0 if unsuccessful.

## ⚠ CAUTION

Be careful resetting the watchdog timer. It may affect the process.

## PLCC\_time\_since\_start\_of\_sweep

T\_INT32 PLCC\_time\_since\_start\_of\_sweep(struct time\_since\_start\_of\_sweep\_rec  
\*pTimeSinceStartofSweepRec);

```
struct time_since_start_of_sweep_rec {  
    T_WORD    time_since_start_of_sweep;  
};
```

## Description

This function is the C interface to service request #9 (Read Sweep Time from Beginning of Sweep). The function will read the time in milliseconds from the beginning of the sweep.

## InParam pTimeSinceStartOfSweepRec

Pointer to structure containing the time since the start of sweep.

## RetVal

The function will return 1 if successful and 0 if unsuccessful.

## PLCC\_nano\_time\_since\_start\_of\_sweep

T\_INT32 PLCC\_nano\_time\_since\_start\_of\_sweep

```
(struct nano_time_since_start_of_sweep_rec  
    *pNanoTimeSinceStartOfSweepRec);
```

```
struct nano_time_since_start_of_sweep_rec{  
    T_DWORD time_since_start_of_sweep;  
};
```

### Description

Read Sweep Time from the Beginning of Sweep in nanosecond units. This service request will get the time in nanoseconds since the start of the sweep.

### InParam pNanoTimeSinceStartOfSweepRec

Pointer to structure containing the time in nanoseconds since the start of sweep.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## PLCC\_read\_folder\_name

T\_INT32 PLCC\_read\_folder\_name(struct read\_folder\_name\_rec \*pReadFolderNameRec);

```
struct read_folder_name {  
    char    folder_name[MAX_FOLDER_NAME_LENGTH]; /* NULL terminated */  
};  
  
#define MAX_FOLDER_NAME_LENGTH 32
```

### Description

This function is the C interface to the PLC similar to service request #10 (Read Folder Name), which only supports a folder name length of 8 characters, including NULL terminator. This function supports 32 characters, which includes one NULL terminator character. This function will return the application folder name as a NULL terminated string.

## OutParam pReadFolderNameRec

Pointer to structure containing the folder name.

## ReturnVal

The function will return 1 if successful and 0 if unsuccessful

## PLCC\_read\_PLC\_ID

T\_INT32 PLCC\_read\_PLC\_ID(struct read\_PLC\_ID\_rec \*pReadPlcIdRec);

```
struct read_PLC_ID_rec {  
    char PLC_ID[8];  
};
```

## Description

This function is based on service request #11 (Read PLC ID). The function returns the name of the PACSystems controller (in ASCII).

## OutParam pReadPlcIdRec

Pointer to structure containing the PLC Id.

## ReturnVal

The function will return 1 if successful and 0 if unsuccessful.

## PLCC\_read\_PLC\_state

T\_INT32 PLCC\_read\_PLC\_state(struct read\_PLC\_state\_rec \*pReadPLCStateRec);

```
struct read_PLC_state_rec {  
    T_WORD    state;  
};  
  
#define RUN_DISABLED    1  
#define RUN_ENABLED    2
```

## Description

This function is based on service request #12 (Read PLC Run State). This function returns the PLC run state (RUN\_DISABLED or RUN\_ENABLED).

## OutParam pReadPlcStateRec

Pointer to structure containing the PLC state.

## ReturnVal

The function will return 1 if successful and 0 if unsuccessful.

## PLCC\_shut\_down\_plc

T\_INT32 PLCC\_shut\_down\_plc(T\_WORD numberOfSweeps);

### Description

This function is the C interface to service request #13 (Shut Down/Stop PLC). The function stops the PLC at the end of the current sweep if numberOfSweeps is equal to 0. All outputs go to their designated default states at the beginning of the next sweep and the “STOPPED by SVC 13” information fault will be logged in the controller fault table. The numberOfSweeps parameter determines the number of full sweeps that should occur before shutting down the PLC. This is normally set to 0.

### InParam numberOfSweeps

Number of full sweeps that should occur before shutting down the PLC. This is normally set to 0.

### RetVal

The function will return 1 if successful, and 0 if unsuccessful.

## PLCC\_mask\_IO\_interrupts

T\_INT32 PLCC\_mask\_IO\_interrupts(struct mask\_IO\_interrupts\_rec

\*pMaskIoInterruptsRec);

struct mask\_IO\_interrupts\_rec {

T\_WORD mask;

T\_WORD memory\_type;

T\_WORD memory\_address;

};

/\* Possible values for the “mask” element \*/

#define MASK 1

#define UNMASK 0

/\* Valid memory types \*/

#define IBIT 70

#define AIMEM 10

All offsets are 1-based: %I1=1, %I2=2, ... %AI1=1, %AI2=2, ...

Discrete offsets and lengths are in bits and must be byte aligned.

1, 9, 17, 25, ... are valid for offsets

2-8, 10-16, 18-24, ... are invalid for offsets

Analog offsets and lengths must be in words.

## Description

This function is the C interface to service request #17 (Mask/Unmask I/O Interrupt). The function will mask or unmask interrupts from an input module according to the value in mask (MASK or UNMASK). The memory\_type parameter specifies the memory type of the input to mask or unmask and can have a value of %I (IBIT) or %AI (AIMEM). The address specified must match a PACSystems input module with maskable channel and interrupts enabled.

## InParam pMaskIoInterruptsRec

Pointer to structure containing mask I/O interrupt information.

## RetVal

This function will return 1 if successful and 0 if unsuccessful.

## PLCC\_mask\_IO\_interrupts\_ext

---

**Note:** Firmware version 3.50 or higher is required for this function.

---

```
struct mask_IO_interrupts_ext_rec{
    T_WORD action;    /* MASK or UNMASK */
    T_WORD memory_type; /* Address of input interrupt trigger */
    T_DWORD memory_offset;
};

extern T_INT32 PLCC_mask_IO_interrupts_ext(struct
    mask_IO_interrupts_ext_rec
/* Possible values for the "action" element */
#define MASK      1
#define UNMASK    0
/* Valid memory type */
#define PLCVAR_MEM 262 (for use with Variables)
#define IBIT      70
#define AIMEM     10
```

All offsets are 1-based: %I1=1, %I2=2, ... %AI1=1, %AI2=2, ...

Discrete offsets and lengths are in bits and must be byte aligned.

1, 9, 17, 25, ... are valid for offsets

2-8, 10-16, 18-24, ... are invalid for offsets

Analog offsets and lengths must be in words.

## Description

This function is based on the MASK\_IO\_INTR function block. It is used to mask or unmask an interrupt from an I/O board.

When the interrupt is masked, the CPU processes the interrupt but does not schedule the associated logic for execution. When the interrupt is unmasked, the CPU processes the interrupt and schedules the associated logic for execution. When the CPU transitions from Stop to Run, the interrupt is unmasked.

This function provides PLC variable access along with reference addresses having 32 bit offset as input. Memory type and offset specify the address of an input interrupt trigger on an input module that supports interrupts. To specify an IO variable as an input to a routine, use the PLC\_VAR\_MEM memory type and the address of the variable record as the offset. For details, see Section PLC\_VAR\_MEM.

## InParam pMaskIoInterruptsExtRec

Pointer to structure containing mask I/O interrupt information.

## RetVal

1 if successful, 0 if unsuccessful, -1 if not supported

## Errno

This function sets Errno if reference memory is out of range. See cpuErrno.h for possible values.

## PLCC\_read\_IO\_override\_status

T\_INT32 PLCC\_read\_IO\_override\_status(struct read\_IO\_override\_status\_rec

\*pReadOverrideStatusRec);

```
struct read_IO_override_status_rec {  
    T_WORD    override_status;  
};
```

```
#define OVERRIDES_SET    1
```

```
#define NO_OVERRIDES_SET 0
```

## Description

This function is the C interface to service request #18 (Read I/O Override Status). The function will return the override\_status (OVERRIDES\_SET, or NO\_OVERRIDES\_SET).

## OutParam pReadIoOverrideStatusRec

Pointer to structure containing override status information.

## RetVal

The function will return 1 if successful and 0 if unsuccessful.

## PLCC\_set\_run\_enable

T\_INT32 PLCC\_set\_run\_enable(struct set\_run\_enable\_rec \*pSetRunEnableRec);

```
struct set_run_enable_rec {  
    T_WORD    enable;  
};
```

```
#define RUN_ENABLED    1
```

```
#define RUN_DISABLED    2
```

## Description

This function is the C interface to service request #19 (Set Run Enable/Disable). The function will set the PLC in either RUN\_ENABLED or RUN\_DISABLED depending on what value was passed in the structure. Use SVCREQ function #19 to permit the ladder program to control the RUN mode of the CPU.

## InParam pSetRunEnableRec

Pointer to structure containing enable run value.

## RetVal

The function will return 1 if successful and 0 if unsuccessful.

## PLCC\_mask\_timed\_interrupts

```
T_INT32 PLCC_mask_timed_interrupts(struct mask_timed_interrupts_rec
                                *pMaskTimedInterruptRec);

struct mask_timed_interrupts_rec {
    T_WORD      action; /* READ_INTERRUPT_MASK or WRITE_INTERRUPT_MASK */
    T_WORD      status; /* if action is READ_INTERRUPT_MASK then this
                        field has MASK or UNMASK as the return value
                        if the action is WRITE_INTERRUPT_MASK then
                        set this field to MASK or UNMASK */
};

/* Possible "action" field values */
#define READ_INTERRUPT_MASK      0
#define WRITE_INTERRUPT_MASK    1

/* Possible "status" field values */
#define MASK                     1
#define UNMASK                   0
```

### Description

This function is the C interface to service request #22 (Mask/Unmask Timed Interrupts). Use this function to mask or unmasked timed interrupts and to read the current mask. When the interrupts are masked, the PLC CPU will not execute any interrupt block that is associated with a timed interrupt. Timed interrupts are masked/unmasked as a group. They cannot be individually masked or unmasked.

To read current mask, set **action** to READ\_INTERRUPT\_MASK.

To change current mask to unmask timed interrupts, set **action** to WRITE\_INTERRUPT\_MASK and **status** to UNMASK.

To change current mask to mask timed interrupts, set **action** to WRITE\_INTERRUPT\_MASK and **status** to MASK.

Successful execution will occur unless some number other than 0 or 1 is entered as the requested operation or mask value.

### In/OutParam pMaskTimedInterruptsRec

Pointer to structure containing masked timed interrupt values.

### ReturnVal

1 if successful

0 if unsuccessful



-1 if not supported

## PLCC\_sus\_res\_HSC\_interrupts

T\_INT32 PLCC\_sus\_res\_HSC\_interrupts(struct sus\_HSC\_interrupts\_rec  
\*pSusResHscInterruptsRec);

```
struct sus_res_HSC_interrupts_rec {
    T_WORD    action; /* SUSPEND or RESUME */
    T_WORD    memory_type;
    T_WORD    reference_address
};

/*Valid memory types */
#define IBIT      70
#define AIMEM     10
```

```
/*Valid "action" values */
#define RESUME     0
#define SUSPEND    1
```

All offsets are 1-based: %I1=1, %I2=2, ... %AI1=1, %AI2=2, ...

Discrete offsets and lengths are in bits and must be byte aligned.

1, 9, 17, 25, ... are valid for offsets

2-8, 10-16, 18-24, ... are invalid for offsets

Analog offsets and lengths must be in words.

## Description

This function is based on service request #32 (Suspend High Speed Counter Interrupts). The function will enable or disable the high speed counter interrupts for a given address and memory type.

## InParam pSusResHscInterruptsRec

Pointer to structure containing high speed counter interrupt suspension/resumption values.

## ReturnVal

1 if successful

0 if unsuccessful

-1 if not supported

## PLCC\_sus\_res\_interrupts\_ext

---

**Note:** Firmware version 3.50 or higher is required for this function.

---

```
struct sus_res_interrupts_ext_rec{
    T_WORD action; /* SUSPEND or RESUME */
    T_WORD memory_type; /* Address of the interrupt trigger */
    T_DWORD memory_offset;
};

extern T_INT32 PLCC_sus_res_interrupts_ext(struct sus_res_interrupts_ext_rec
*pSusResInterruptsExtRec);
```

*/\* Possible values for the "action" element \*/*

```
#define SUSPEND      1
```

```
#define RESUME        0
```

*/\* Valid memory type \*/*

```
#define PLCVAR_MEM 262 (for use with Variables)
```

```
#define IBIT    70
```

```
#define AIMEM    10
```

All offsets are 1-based: %I1=1, %I2=2, ... %AI1=1, %AI2=2, ...

Discrete offsets and lengths are in bits and must be byte aligned.

1, 9, 17, 25, ... are valid for offsets

2-8, 10-16, 18-24, ... are invalid for offsets

Analog offsets and lengths must be in words.

### Description

This function is based on the SUSP\_IO\_INTR function block. It is used to suspend or resume an I/O interrupt. Currently it is supported only for High Speed Counter.

This function provides PLC variable access along with reference addresses having 32-bit offset as input. Memory type and offset specify the address of an input interrupt trigger on an input module that supports interrupts. To specify a PLC variable as an input to a routine, use the PLC\_VAR\_MEM memory type and the address of the variable record as the offset. For details, see Section PLC\_VAR\_MEM.

When used for reference addresses, all offsets are 1-based: %I1=1, %I2=2, ... %AI1=1, %AI2=2, ...

Discrete offsets and lengths are in bits and must be byte aligned.

1, 9, 17, 25, ... are valid for offsets

2-8, 10-16, 18-24, ... are invalid for offsets

Analog offsets and lengths must be in words.

### InParam pSusResInterruptsExtRec

A pointer to Suspend Resume Interrupts Extn record.

### ReturnVal

1 if successful

0 if unsuccessful

-1 if not supported

### Errno

This function sets Errno if reference memory is out of range. See cpuErrno.h for possible values.

## 3.5.5.1 PLCC\_get\_escm\_status

INT32 PLCC\_get\_escm\_status (struct escm\_status\_rec \*pEscmStatusRec);

```
struct escm_status_rec {  
    T_WORD port_number;  
    T_WORD port_status;
```

```
};
```

```
#define port_1      1
```

```
#define port_2      2
```

### Description

If the function return value is zero (0), the function was not successful, usually indicating that the PLC does not support ESCM (embedded serial communications module) ports (see Note below). If the function return value is one (1), the function was successful.

This function also returns a status word for Ports 1 or 2 (word port\_status). The bit values for that word are shown in the following table:

## Port\_Status for the PLCC\_get\_escm\_status Function

Port Status	Meaning
bit 0	PORTN_OK: Requested port is ready. If value is 1, the port is ready. If value is 0, the port is not usable.
bit 1	PORTN_ACTIVE: There is activity on this port. If value is 1, the port is active. If value is 0, the port is inactive.
bit 2	PORTN_DISABLED: Requested port is disabled. If value is 1, the port is disabled. If value is 0, the port is enabled.
bit 3	PORTN_FUSE_BLOWN: Requested port's fuse is blown (for Port 2) or supply voltage is not within range (for Port 1). If value is 1, the fuse is blown (or voltage not within range). If value is 0, the fuse (or supply voltage) is okay.

**Note:** Because the ESCM is not supported on the PACSystems CPUs, this function always returns a value of 0.

## OutParam pEscmStatusRec

A pointer to an escm\_status\_rec.

## ReturnVal

1 if successful

0 if unsuccessful or ESCM is not supported.

## PLCC\_set\_application\_redundancy\_mode

**Note:** CPU firmware version 5.00 or higher is required for this function.

```
extern T_INT32 PLCC_set_application_redundancy_mode(T_WORD mode);
```

```
/* Possible values for the redundancy mode. */
```

```
#define BACKUP_MODE 0
```

```
#define ACTIVE_MODE 1
```

## Description

**Note:** The PLCC\_set\_application\_redundancy\_mode function is recognized only in non-HSB (hot standby) CPUs. (These CPUs have a "CPE" or "CPU" designation.)

This function is intended for use in user-developed redundancy applications. In these systems, the application logic coordinates between CPUs that act as redundant partners, and determines which CPU is the active unit and which are backup units. This function is not needed for HSB (CRE) CPUs, because the redundancy firmware in those CPUs automatically adjusts the active/backup role of each Ethernet interface that is configured for redundant IP operation.

This service request sends a role switch command to all Ethernet interfaces in the PLC that are configured for redundant IP operation. When a redundancy role switch occurs, the backup CPU becomes active and begins responding to the Redundant IP address in addition to its direct IP address. The formerly active CPU switches to backup and stops communicating on the network using the Redundant IP address.

PLCC\_set\_application\_redundancy\_mode has no effect on Ethernet interfaces that are not configured for redundant IP operation.

For information on Ethernet redundancy operation, refer to the PACSystems RX7i, RX3i and RSTi-EP TCP/IP Ethernet Communications User Manual, GFK 2224.

## InParam mode

The requested redundancy mode: Use 0 for backup mode, or 1 for active mode.

## ReturnVal

This function will return 1 if successful and 0 if unsuccessful.

## 3.5.6 Fault Table Service Request Functions

The following functions access the fault table. These functions are defined in ctkPlcFault.h.

The following definitions and structures are common to the Fault Table Service Request Functions:

```
#define NUM_LEGACY_PLC_FAULT_ENTRIES    16
#define NUM_LEGACY_IO_FAULT_ENTRIES    32

#define PLC_FAULT_TABLE                0
#define IO_FAULT_TABLE                 1
#define IO_REMOTE_FAULT_TABLE          0x41

#define PLC_EXT_FAULT_TABLE             0x80
#define IO_EXT_FAULT_TABLE              0x81
#define IO_EXT_REMOTE_FAULT_TABLE       0xC1

/*
 * NOTE: time stamps are in BCD format
 */
struct time_stamp_rec{
    T_BYTE second;    /* BCD format, seconds in low-order nibble, */
                      /* tens of seconds in high-order nibble. */
    T_BYTE minute;    /* BCD format, same as for seconds. */
    T_BYTE hour;      /* BCD format, same as for seconds. */
    T_BYTE day;       /* BCD format, same as for seconds. */
}
```

```

    T_BYTE month;      /* BCD format, same as for seconds.      */
    T_BYTE year; /* BCD format, same as for seconds.      */
};

struct ext_time_stamp_rec{
    T_BYTE second;      /* BCD format, seconds in low-order nibble,      */
                        /* /* tens of seconds in high-order nibble.      */
    T_BYTE minute;      /* BCD format, same as for seconds.      */
    T_BYTE hour;        /* BCD format, same as for seconds.      */
    T_BYTE day; /* BCD format, same as for seconds.      */
    T_BYTE month;      /* BCD format, same as for seconds.      */
    T_BYTE year; /* BCD format, same as for seconds.      */
    T_WORD millisecond; /* BCD format, 0HTO ms format, milliseconds */
                        /* /* in low-order nibble (xxxO), tens (xxTx), hundreds next (xHxx). */
};

struct PLC_flt_address_rec{
    T_BYTE rack;
    T_BYTE slot;
    T_WORD task;
};

struct IO_flt_address_rec{
    T_BYTE rack;
    T_BYTE slot;
    T_BYTE IO_bus;
    T_BYTE block;
    T_WORD point;
};

struct IO_flt_remote_address_rec{
    T_BYTE rack;
    T_BYTE slot;
    T_BYTE remote_rack_num;
    T_BYTE remote_slot_num;
    T_BYTE Device_id;
    T_BYTE sub_slot;
};

```

```
        T_BYTE IO_bus;
        T_BYTE block;
        T_WORD point;
};

struct reference_address_rec{
    T_BYTE memory_type;
    T_WORD offset;
};

/* Note: this is the long PLC fault entry type */
struct PLC_fault_entry_rec{
    T_BYTE long_short;
    T_BYTE reserved[3];
    struct PLC_fit_address_rec PLC_fault_address;
    T_BYTE fault_group;
    T_BYTE fault_action;
    T_WORD error_code;
    T_WORD fault_specific_data[12];
    struct time_stamp_rec time_stamp;
};

struct IO_fault_entry_rec{
    T_BYTE long_short;
    struct reference_address_rec reference_address;
    struct IO_fit_address_rec IO_fault_address;
    T_BYTE fault_group;
    T_BYTE fault_action;
    T_BYTE fault_category;
    T_BYTE fault_type;
    T_BYTE fault_description;
    T_BYTE fault_specific_data[21];
    struct time_stamp_rec time_stamp;
};

struct IO_remote_fault_entry_rec{
    T_BYTE long_short;
```

```
struct reference_address_rec reference_address;
struct IOflt_remote_address_rec IO_fault_address;
T_BYTE fault_group;
T_BYTE fault_action;
T_BYTE fault_category;
T_BYTE fault_type;
T_BYTE fault_description;
T_BYTE fault_specific_data[21];
struct time_stamp_rec time_stamp;
};
```

```
struct PLC_ext_fault_entry_rec{
    T_BYTE long_short;
    T_BYTE reserved[3];
    struct PLCflt_address_rec PLC_fault_address;
    T_BYTE fault_group;
    T_BYTE fault_action;
    T_WORD error_code;
    T_WORD fault_specific_data[12];
    struct ext_time_stamp_rec time_stamp;
    T_WORD fault_id;
};
```

```
struct IO_ext_fault_entry_rec{
    T_BYTE long_short;
    struct reference_address_rec reference_address;
    struct IOflt_address_rec IO_fault_address;
    T_BYTE fault_group;
    T_BYTE fault_action;
    T_BYTE fault_category;
    T_BYTE fault_type;
    T_BYTE fault_description;
    T_BYTE fault_specific_data[21];
    struct ext_time_stamp_rec time_stamp;
```



```

    T_WORD fault_id;
};
struct IO_ext_remote_fault_entry_rec{
    T_BYTE long_short;
    struct reference_address_rec reference_address;
    struct IO_fit_remote_address_rec IO_fault_address;
    T_BYTE fault_group;
    T_BYTE fault_action;
    T_BYTE fault_category;
    T_BYTE fault_type;
    T_BYTE fault_description;
    T_BYTE fault_specific_data[21];
    struct ext_time_stamp_rec time_stamp;
    T_WORD fault_id;
};

```

## PLCC\_clear\_fault\_tables

```
T_INT32 PLCC_clear_fault_tables(struct clear_fault_tables_rec *x);
```

```

struct clear_fault_tables_rec {
    T_WORD table;
};

```

```
/* Valid "table" values */
```

```
#define PLC_FAULT_TABLE    0
```

```
#define IO_FAULT_TABLE     1
```

## Description

This function is the C interface to service request #14 (Clear Fault Tables). The function will clear the fault table according to the value (PLC\_FAULT\_TABLE or IO\_FAULT\_TABLE).

## InParam x

Pointer to structure which indicates whether to clear the PLC or the I/O fault table.

## RetVal

The function returns 1 if successful and 0 if unsuccessful.

## PLCC\_read\_last\_fault

INT32 PLCC\_read\_last\_fault(struct read\_last\_fault\_rec \*x);

```
struct read_last_fault_rec {  
    T_WORD      table;  
    union {  
        struct      PLC_entry_rec  PLC_entry;  
        struct      IO_entry_rec   IO_entry_rec;  
    } faultEntry; /* Note: 90-70 C Toolkit did not require union name */  
};  
/* Valid "table" values */  
#define PLC_FAULT_TABLE    0  
#define IO_FAULT_TABLE    1
```

## Description

This function is the C interface to service request #15 (Read Last-Logged Fault Table Entry). The function will return the last fault table entry of the table specified in the table field (PLC\_FAULT\_TABLE, or IO\_FAULT\_TABLE).

In the return data, the long/short indicator defines the quantity of fault data present in the fault entry. In the controller fault table, a long/short value of 00 represents 8 bytes of fault extra data present in the fault entry, and 01 represents 24 bytes of fault extra data. In the I/O fault table, 02 represents 5 bytes of fault specific data, and 03 represents 21 bytes.

## InParam x

Pointer to structure containing record of last PLC and I/O fault.

## Return Data

The function returns a 1 if successful and a 0 if unsuccessful.

## PLCC\_read\_fault\_tables

```
T_INT32 PLCC_read_fault_tables(struct read_fault_tables_rec *x);

struct read_fault_tables_rec {
    T_WORD    table;
    T_WORD    zero;
    T_WORD    reserved[13];
    struct    time_stamp_rec time_since_clear;
    T_WORD    num_faults_since_clear;
    T_WORD    num_faults_in_queue;
    T_WORD    num_faults_read;
    union {
        struct PLC_entry_rec  PLC_faults[NUM_LEGACY_PLC_FAULT_ENTRIES];
        struct IO_entry_rec   IO_faults[NUM_LEGACY_IO_FAULT_ENTRIES];
        struct IO_remote_fault_entry_rec
        IO_remote_faults[NUM_LEGACY_IO_FAULT_ENTRIES];

        }faultEntry; /* 90-70 C Toolkit did not require union name */
};

#define PLC_FAULT_TABLE      0
#define IO_FAULT_TABLE      1
#define IO_REMOTE_FAULT_TABLE 0x41
```

## Description

This function is the C interface to service request #20 (Read Fault Tables). The function will read the fault table specified in the table field (PLC\_FAULT\_TABLE or IO\_FAULT\_TABLE). The function will return the table in an array of PLC\_faults or IO\_faults. The zero field and the reserved fields do not hold fault data. The **time\_since\_clear** fields are BCD numbers with seconds in the low order nibble and tens of seconds in the high order nibble. The **num\_faults\_since\_clear** field shows the number of faults that have occurred since the table was last cleared. The **num\_faults\_read** field shows the number of faults read into the arrays for I/O and PLC faults; there is room for the entire table, but only the **num\_faults\_read** field will have valid data.

In the return data, the long/short indicator defines the quantity of fault data present in the fault entry. In the controller fault table, a long/short value of 00 represents 8 bytes of fault extra data present in the fault entry, and 01 represents 24 bytes of fault extra data. In the I/O fault table, 02 represents 5 bytes of fault specific data, and 03 represents 21 bytes.

This function provides a maximum of 16 controller fault table entries and 32 I/O fault table entries. If the fault table read is empty, no data is returned.

### InParam x

Pointer to structure containing record of all current PLC or I/O fault table entries.

### Return Data

The function will return 1 if successful, and 0 if unsuccessful.

## PLCC\_read\_last\_ext\_fault

```
T_INT32 PLCC_read_last_ext_fault(struct read_last_ext_fault_rec *x);
```

```
struct read_last_ext_fault_rec {  
    T_WORD table; /* PLC_EXT_FAULT_TABLE or IO_EXT_FAULT_TABLE */  
    union {  
        struct PLC_ext_fault_entry_rec PLC_entry;  
        struct IO_ext_fault_entry_rec IO_entry; } faultEntry; /* note: 90-70 C Toolkit did not  
require name for union */  
};
```

```
/* Use the following definitions for "table" */
```

```
#define PLC_EXT_FAULT_TABLE 0x80
```

```
#define IO_EXT_FAULT_TABLE 0x81
```

### Description

This service request will read the last entry logged in either the PLC or I/O fault table with the extended format. This function is the C interface to service request #15 when the fault table entry value is either PLC\_EXT\_FAULT\_TABLE or IO\_EXT\_FAULT\_TABLE.

### InParam x

Pointer to structure containing extended record of last PLC and I/O fault.

### ReturnVal

1 if successful

0 if unsuccessful

-1 if not supported

## PLCC\_read\_ext\_fault\_tables

```
T_INT32 PLCC_read_ext_fault_tables(struct read_ext_fault_tables_rec *x);
```

```
struct read_ext_fault_tables_rec {  
    T_WORD table; /* PLC_EXT_FAULT_TABLE or IO_EXT_FAULT_TABLE or  
IO_EXT_REMOTE_FAULT_TABLE */  
    T_WORD start_index;  
    T_WORD number_of_entries_to_read;  
    T_WORD reserved[12];  
    struct time_stamp_rec time_since_clear;  
    T_WORD num_faults_since_clear;  
    T_WORD num_faults_in_queue;  
    T_WORD num_faults_read;  
    T_WORD PlcName[16];  
    union{  
        struct PLC_ext_fault_entry_rec PLC_faults[1];  
        struct IO_ext_fault_entry_rec IO_faults[1];  
        struct IO_ext_remote_fault_entry_rec IO_remote_faults[1];  
    } faultEntry; /* note: 90-70 C Toolkit did not require name for union */  
};  
  
/* Use the following definitions for "table" */  
#define PLC_EXT_FAULT_TABLE      0x80  
#define IO_EXT_FAULT_TABLE      0x81  
#define IO_EXT_REMOTE_FAULT_TABLE 0xC1  
  
/* Note the faultEntry member structures are intended to be variable size  
arrays. See Appendix A for instructions on how to change the size of the  
array.*/
```

### Description

This service request will read the entire PLC or I/O fault table in extended

format. This function is the C interface to service request #20 (Read Fault Tables) when the table is specified to be either PLC\_EXT\_FAULT\_TABLE or IO\_EXT\_FAULT\_TABLE.

### InParam x

Pointer to structure containing record of all PLC or I/O fault tables in extended format.

### RetVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## 3.5.7 Module Communications

### PLCC\_comm\_req

T\_INT32 PLCC\_comm\_req(struct comm\_req\_rec \*pCommReqRec);

```
struct status_addr {
    T_WORD    seg_selector;
    T_WORD    offset;
};

struct comm_req_command_blk_rec {
    T_WORD    length;
    T_WORD    wait;
    struct    status_addr    status;
    T_WORD    idle_timeout;
    T_WORD    max_comm_time;
    T_WORD    data[128];
};

struct comm_req_rec {
    struct    comm_req_command_blk_rec *command_blk;
    T_BYTE    slot;
    T_BYTE    rack;
    T_DWORD    task_id;
    T_BYTE    ft;    /*ft is set if the COMMREQ fails */
};
```

## Description

This function is based on the COMMREQ ladder logic block.

## InParam pCommReqRec

A pointer to communications request record.

## RetVal

The function returns 1 if successful and 0 if unsuccessful.

## 3.5.8 Ladder Function Blocks

### PLCC\_do\_io

T\_INT PLCC\_do\_io(struct do\_io\_rec \* pDoloRec);

```
struct do_io_rec {
    T_BYTE      start_mem_type;
    T_WORD      start_mem_offset;
    T_WORD      length;
    T_BYTE alt_mem_type; /* must be set to NULL_SEGSEL if not used */
    T_WORD      alt_mem_offset;
};

#define NULL_SEGSEL      0xFF (Only valid for alt_mem_type)

/* Valid memory types */
#define I_MEM            16
#define Q_MEM            18
#define R_MEM            8
#define AI_MEM           10
#define AQ_MEM           12
#define W_MEM            196
```

## Description

This function is used to update inputs or outputs for one scan while the program is running. This function can be used in conjunction with the Suspend I/O function (Section PLCC\_sus\_io), which stops the normal I/O scan. It can also be used to update selected I/O during the program, in addition to the normal I/O scan.

If input references are specified, the function allows the most recent values of inputs to be obtained for program logic. If output references are specified, **PLCC\_do\_io** updates

outputs based on the most current values stored in I/O memory. I/O points are serviced in increments of entire I/O modules; the PLC adjusts the references, if necessary, while the function executes. The **PLCC\_do\_io** function will not scan I/O modules that are not configured.

The **PLCC\_do\_io** function is supported for most PACSystems modules. It does not support Genius I/O modules. The **PLCC\_do\_io** function skips modules that do not support DO\_IO scanning. For details, see “Control Functions” in the PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual, GFK 2222.

When this function executes, the input point specified by **start\_mem\_type** and **start\_mem\_offset** and the bits included (as specified by length) are scanned. If **alternate\_mem\_type** and **alternate\_mem\_offset** is defined, a copy of the data is placed in alternate memory, and the real input points are not updated. If this function references output data, data specified in **start\_mem\_type** and **start\_mem\_offset** is written to the output modules. If alt locations are defined, the alternate data is written to the output modules.

Execution of the function continues until either all inputs in the selected range have reported or all outputs have been serviced on the I/O cards.

For **PLCC\_do\_io**, the Offset and Length for Word types is in units of Words. For Bit types, the Offset and Length is in units of Bits. Offset and Length is 1-based.

## InParam pDoloRec

A pointer to Do I/O record.

## RetVal

The function returns a 1 unless one or more of the following is true (in which case it returns a 0):

- Not all references of the type specified are present within the selected range.
- The CPU is not able to properly handle the temporary list of I/O created by the function.
- The range specified includes I/O modules that are associated with a Loss of I/O Module fault.

---

**Note:** *If the function is used with timed or I/O interrupts, transitional contacts associated with scanned inputs may not operate as expected. If an I/O or Alt reference address, including length, is outside the configured memory limits, the function will set errno with values described in CPUErrno.h.*

---



## PLCC\_do\_io\_ext

---

**Note:** Firmware version 3.50 or higher is required for this function.

---

```
struct do_io_ext_rec{
    T_WORD start_mem_type;
    T_DWORD start_mem_offset;
    T_DWORD length; /* Ignored if start_mem_type is PLC_VAR_MEM */
    T_WORD alt_mem_type; /* must be set to NULL_SEGSEL if not used */
    T_DWORD alt_mem_offset;
};

/* Valid memory types */
#define I_MEM          16
#define Q_MEM          18
#define R_MEM          8
#define AI_MEM         10
#define AQ_MEM         12
#define W_MEM          196
#define PLCVAR_MEM 262

extern T_INT32 PLCC_do_io_ext(struct do_io_ext_rec *pDoloExtRec);
```

### Description

This function is an extension of PLCC\_do\_io. It is used to update inputs or outputs for one scan while the program is running. This function can be used in conjunction with the Suspend I/O function (Section PLCC\_sus\_io), which stops the normal I/O scan. It can also be used to update selected I/O during the program, in addition to the normal I/O scan.

This function provides PLC variable access along with reference addresses having 32-bit offset as input. To specify a PLC variable as an input to a routine, use the PLC\_VAR\_MEM memory type and the address of the variable record as the offset. For details, see Section PLC\_VAR\_MEM.

### InParam pDoloRec

A pointer to the Do I/O Extn record.

### ReturnVal

1 if successful  
0 if unsuccessful  
-1 if not supported

## Errno

Sets Errno if input memory or alt memory is out of range. See `cpuErrno.h` for possible values.

## PLCC\_sus\_io

`T_INT32 PLCC_sus_io(void);`

### Description

This function is used to stop normal I/O scans from occurring for **one** CPU sweep. During the next output scan, all outputs are held at their current states. During the next input scan, the input references are not updated with data from inputs. However, during the input scan portion of the sweep the CPU will verify that Genius Bus Controllers have completed their previous output updates.

---

**Note:** *This function suspends all I/O, both analog and discrete, whether rack I/O or Genius I/O.*

---

### RetVal

The `PLCC_sus_io` function returns a 1 if successful, 0 if unsuccessful.

## PLCC\_scan\_set\_io

---

**Note:** *CPU firmware version 5.00 or higher is required for this function.*

---

```
struct scan_set_io_rec{
    T_BOOLEAN scan_inputs;
    T_BOOLEAN scan_outputs;
    T_UINT16 scan_set_number;
};
```

```
extern T_INT32 PLCC_scan_set_io(struct scan_set_io_rec *pScanSetIoRec);
```

### Description

This function scans the I/O of a specified scan set number. (Modules can be assigned to scan sets in hardware configuration.) You can specify whether the Inputs and/or Outputs of the associated scan set will be scanned.

Execution of this function block does not affect the normal scanning process of the corresponding scan set. If the corresponding scan set is configured for non-default Number of Sweeps or Output Delay settings, they remain in effect regardless of how many executions of the Scan Set IO function occur in any given sweep.

The Scan Set IO function skips modules that do not support DO\_IO scanning. For details, see Control Functions in the PACSystems RX7i, RX3i and RSTi-EP CPU Reference Manual, GFK 2222.

### InParam pScanSetIo

A pointer to Scan Set IO record.

### RetVal

The `PLCC_scan_set_io` function returns one of the following values:

1 if successful

0 if unsuccessful

-1 if not supported

## 3.5.9 Miscellaneous General Functions

The following miscellaneous functions are described in `ctkPlcFunc.h`.

### PLCC\_SNP\_ID

`T_INT PLCC_SNP_ID (T_BYTE request_type, char id_str_ptr);`

`/* Valid "request_type" values */#define READ_ID 0`

`#define WRITE_ID 1`

### Description

This function will read or write the SNP ID string passed in through `id_str_ptr` to the PLC. The string should be an eight-character buffer (space for seven letters and a NULL termination).

### InParam request\_type

Indicates whether the SNP Id should be read or written.

### InParam id\_str\_ptr

Pointer to character buffer that contains the id to write or receives the current id. This buffer needs to be allocated by the caller.

### RetVal

This function returns 1 if successful, 0 if unsuccessful, and -1 if unsupported.

## PLCC\_read\_override

T\_INT32 PLCC\_read\_override (BYTE seg\_sel, WORD ref\_num, WORD len,  
BYTE \*data);

```
/* Valid "seg_sel" values */
```

```
#define I_OVR      I_MEM /* this was 0 for the 90-70 C Toolkit */
```

```
#define Q_OVR      Q_MEM /* this was 1 for the 90-70 C Toolkit */
```

```
#define M_OVR      M_MEM /* this was 2 for the 90-70 C Toolkit */
```

```
#define G_OVR      G_MEM /* this was 3 for the 90-70 C Toolkit */
```

### Description

This function reads the override table for the specified type. The read at the offset must be byte-aligned, that is, ref\_num must be set to a value from the following series 1, 9, 17, 33,... The length is in bytes. The area pointed to by data must be large enough to hold the amount being read.

### InParam seg\_sel

Indicates the segment selector of the table to get the override values. For example, use %I segment selector to access the override table associated with %I.

### InParam ref\_num

Indicates which reference number to start reading from the override table. The address should be byte aligned for discrete memory (1, 9, 17 etc).

### InParam len

Indicates the number of bytes to read from the override table starting from ref\_num

### OutParam data

Pointer to memory location to put the requested override data.

### ReturnVal

This function returns:

- 0 if successful
- 2 bad\_memory\_type
- 3 offset\_not\_byte\_aligned
- 4 reading\_outside\_ref\_mem
- 5 bad\_data\_pointer

## 3.5.10 Reference Memory Functions

The functions in this section are used to access PLC reference memory. These functions properly take into account transitions and overrides. In addition, they perform memory range checking. These functions are described in `ctkRefMem.h`. When specifying the “Ref Table” input parameter, use the following values:

Ref Table Name	Ref Table ID Value
R_MEM	8
AI_MEM	10
AQ_MEM	12
W_MEM	196
I_MEM	16
Q_MEM	18
T_MEM	20
M_MEM	22
SA_MEM	24
SB_MEM	26
SC_MEM	28
S_MEM	30
AI_DIAG_MEM	40
AQ_DIAG_MEM	42
G_MEM	56
I_BIT	70
I_DIAG_MEM	110
Q_DIAG_MEM	112
I_TRANS_MEM	132
Q_TRANS_MEM	134
T_TRANS_MEM	136
M_TRANS_MEM	138
SA_TRANS_MEM	140
SB_TRANS	142
SC_TRANS_MEM	144
S_TRANS_MEM	146
G_TRANS_MEM	148

RPT_FLT_MEM	188
NULL_SEGSEL	0xff
PLC_VAR_MEM	262

## PLC\_VAR\_MEM

PLC\_VAR\_MEM is used for PLC variable access. When PLC\_VAR\_MEM is used, the offset should be the address of the PLC variable record. This memory type must be used on a routine that supports a 32-bit offset.

For example:

```
mask_io_intr_ext_rec.action = MASK;
mask_io_intr_ext_rec.memory_type = PLC_VAR_MEM;
mask_io_intr_ext_rec.memory_offset = &myVarRec;
```

The variables used must be internally or externally published in the PLC. If they are not published, store to the PLC will fail.

## WritePlcByte

T\_INT32 WritePlcByte(T\_WORD RefTable, T\_DWORD offset, T\_BYTE writeValue,  
T\_BOOLEAN msbByte);

### Description

This function writes to reference memory taking into account overrides and transition bits. A byte of reference memory in the specified Reference Table (RefTable) and at the specified "offset" is written with the "writeValue".

### InParam RefTable

Reference table to write.

### InParam offset

Offset within the reference table to write.

---

**Note:** the offset is 1-based.

---

For example, RefTable = R\_MEM and offset = 1 accesses %R00001.

### InParam writeValue

The value to write to the specified reference table and offset.

### InParam msbByte

For word references, determines whether the byte is written to the most significant byte (msbByte = TRUE) or to the least significant byte (msbByte = FALSE).

## RetVal

If the "RefTable" or "offset" are out of range, no reference memory values are changed, and the function returns GEF\_ERROR. If the "offset" is within range, the function returns GEF\_OK.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## ReadPlcByte

T\_BYTE ReadPlcByte (T\_WORD RefTable, T\_DWORD offset, T\_BOOLEAN msbByte);

### Description

A byte of reference memory in the specified Reference Table (RefTable) and at the specified "offset" is read and returned by the function. Errno is set if there is an error reading the value.

### InParam RefTable

Reference table to read.

### InParam offset

Offset within the reference table to read.

---

**Note:** The offset is 1-based.

---

For example, RefTable = R\_MEM and offset = 1 accesses %R00001

### InParam msbByte

For word references, determines whether the byte is read from the most significant byte (msbByte = TRUE) or to the least significant byte (msbByte = FALSE).

## RetVal

The value read from the specified reference table at the specified offset.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## WritePlcWord

T\_INT32 WritePlcWord(T\_WORD RefTable, T\_DWORD offset, T\_WORD writeValue);

### Description

This function writes to reference memory taking into account overrides and transition bits. A word (16 unsigned bits) of reference memory in the specified Reference Table (RefTable) and at the specified "offset" is written with the "writeValue".

### InParam RefTable

Reference table to write.

### InParam offset

Offset within the reference table to write.

---

**Note:** The offset is 1-based.

---

For example, RefTable = R\_MEM and offset = 1 accesses %R00001

### InParam writeValue

The value to write to the specified reference table and offset

### ReturnVal

If the "RefTable" or "offset" are out of range, no reference memory values are changed, and the function returns GEF\_ERROR. If the "offset" is within range, the function returns GEF\_OK.

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## ReadPlcWord

T\_WORD ReadPlcWord (T\_WORD RefTable, T\_DWORD offset);

### Description

A word (16 unsigned bits) of reference memory in the specified Reference Table (RefTable) and at the specified offset is read and returned by the function. Errno is set if there is an error reading the value.

### InParam RefTable

Reference table to read.

### InParam offset

Offset within the reference table to read.

---

**Note:** The offset is 1-based.

---



For example, RefTable = R\_MEM and offset = 1 accesses %R00001

## ReturnVal

The value read from the specified reference table at the specified offset

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## WritePlcInt

T\_INT32 WritePlcInt(T\_WORD RefTable, T\_DWORD offset, T\_INT16 writeValue);

## Description

This function writes to reference memory taking into account overrides and transition bits. Reference memory in the specified Reference Table (RefTable) and at the specified "offset" is written with the "writeValue" as a 16-bit signed integer.

## InParam RefTable

Reference table to write.

## InParam offset

Offset within the reference table to write.

---

**Note:** The offset is 1-based.

---

For example, RefTable = R\_MEM and offset = 1 accesses %R00001

## InParam writeValue

The value to write to the specified reference table and offset

## ReturnVal

If the "RefTable" or "offset" are out of range, no reference memory values are changed, and the function returns GEF\_ERROR. If the "offset" is within range, the function returns GEF\_OK.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

## ReadPlcInt

`T_INT16 ReadPlcInt (T_WORD RefTable, T_DWORD offset);`

### Description

Reference memory in the specified Reference Table (RefTable) and at the specified "offset" is read as a 16-bit signed integer and returned by the function. Errno is set if there is an error reading the value.

### InParam RefTable

Reference table to read.

### InParam offset

Offset within the reference table to read.

---

**Note:** The offset is 1-based.

---

For example, RefTable = `R_MEM` and offset = 1 accesses `%R00001`

## ReturnVal

The value read from the specified reference table at the specified offset

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

## WritePlcDint

`T_INT32 WritePlcDint (T_WORD RefTable, T_DWORD offset, T_INT32 writeValue);`

### Description

This function writes to reference memory taking into account overrides and transition bits. Reference memory in the specified Reference Table (RefTable) and at the specified "offset" is written with the "writeValue" as a 32-bit signed integer.

### InParam RefTable

Reference table to write.

### InParam offset

Offset within the reference table to write.

---

**Note:** *The offset is 1-based.*

---

For example, RefTable = R\_MEM and offset = 1 accesses %R00001

## InParam writeValue

The value to write to the specified reference table and offset

## ReturnVal

If the "RefTable" or "offset" are out of range, no reference memory values are changed, and the function returns GEF\_ERROR. If the "offset" is within range, the function returns GEF\_OK.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## ReadPlcDint

T\_INT32 ReadPlcDint (T\_WORD RefTable, T\_DWORD offset);

## Description

Reference memory in the specified Reference Table (RefTable) and at the specified offset is read as a 32-bit signed integer and returned by the function. Errno is set if there is an error reading the value.

## InParam RefTable

Reference table to read.

## InParam offset

Offset within the reference table to read.

---

**Note:** *The offset is 1-based.*

---

For example, RefTable = R\_MEM and offset = 1 accesses %R00001

## ReturnVal

The value read from the specified reference table at the specified offset.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

## WritePlcDouble

`T_INT32 WritePlcDouble (T_WORD RefTable, T_DWORD offset, T_REAL64 writeValue);`

### Description

This function writes to reference memory taking into account overrides and transition bits. Reference memory in the specified Reference Table (RefTable) and at the specified “offset” is written with the “writeValue” as a 64-bit floating point value.

### InParam RefTable

Reference table to write.

### InParam offset

Offset within the reference table to write.

---

**Note:** *The offset is 1-based.*

---

For example, RefTable= R\_MEM and offset = 1 accesses %R00001.

### InParam writeValue

The value to write to the specified reference table and offset.

### ReturnVal

If the “RefTable” or “offset” are out of range, no reference memory values are changed, and the function returns `GEF_ERROR`. If the “offset” is within range, the function returns `GEF_OK`.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

## ReadPlcDouble

T\_REAL64 ReadPlcDouble (T\_WORD RefTable, T\_WORD offset);

### Description

Reference memory in the specified Reference Table (RefTable) and at the specified offset is read as a 64-bit floating point value and returned by the function. Errno is set if there is an error reading the value.

### InParam RefTable

Reference table to read.

### InParam offset

Offset within the reference table to read.

---

**Note:** The offset is 1-based.

---

For example, RefTable= R\_MEM and offset = 1 accesses %R00001.

### RetVal

The value read from the specified reference table at the specified offset.

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## WritePlcFloat

T\_INT32 WritePlcFloat (T\_WORD RefTable, T\_DWORD offset, T\_REAL32 writeValue);

### Description

This function writes to reference memory taking into account overrides and transition bits. Reference memory in the specified Reference Table (RefTable) and at the specified “offset” is written with the “writeValue” as a 32-bit floating point value.

### InParam RefTable

Reference table to write.

### InParam offset

Offset within the reference table to write.

---

**Note:** The offset is 1-based.

---

For example, RefTable= R\_MEM and offset = 1 accesses %R00001.

### InParam writeValue

The value to write to the specified reference table and offset.

## RetVal

If the “RefTable” or “offset” are out of range, no reference memory values are changed, and the function returns GEF\_ERROR. If the “offset” is within range, the function returns GEF\_OK.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

---

**Note:** Only RSTi-EPSCPE1xx platform supports this function.

---

## ReadPlcFloat

T\_REAL32 ReadPlcDouble (T\_WORD RefTable, T\_WORD offset);

### Description

Reference memory in the specified Reference Table (RefTable) and at the specified offset is read as a 32-bit floating point value and returned by the function. Errno is set if there is an error reading the value.

### InParam RefTable

Reference table to read.

### InParam offset

Offset within the reference table to read.

---

**Note:** The offset is 1-based.

---

For example, RefTable= R\_MEM and offset = 1 accesses %R00001.

## RetVal

The value read from the specified reference table at the specified offset.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

---

**Note:** Only RSTi-EPSCPE1xx platform supports this function.

---

## PlcMemCopy

T\_INT32 PlcMemCopy (void \*pDestination, void \*pSource, T\_DWORD size);

### Description

This function copies values from one PLC memory location to another, taking into account overrides and transition bits if the destination address is in one of the discrete memory tables. The length of data written is determined by the "size" parameter, which is in units of bytes (8 bits).

### InParam pDestination

Pointer to a PLC memory location to be written.

### InParam pSource

Pointer to PLC memory to be copied into pDestination memory.

### InParam size

Indicates the number of bytes to copy.

### ReturnVal

If one of the pointers to memory is a null pointer, the function returns GEF\_ERROR. In addition, if the source or destination is a reference table and the "size" causes the copy operation to go outside the boundaries of the specified table, the function also returns GEF\_ERROR. If the write operation is successful, the function returns GEF\_OK.

### Errno

If there is an error, this function sets Errno to give more specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## refMemSize

T\_DWORD refMemSize(T\_WORD RefTable);

### Description

This function returns the size of specified reference memory.

### InParam RefTable

Reference table segment selector used to indicate which table to find the size.

## RetVal

Returns the size of reference memory in word units for word type memories and bits for bit type memories and in bytes for analog diagnostic memory.

If RefTable is invalid or pRefLocalSegSizeTable pointer is null, the function returns 0. The function can also return 0 if the memory has been configured with a 0 length.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## setBit

T\_INT32 setBit(T\_WORD RefTable, T\_DWORD offset, T\_WORD bitNumber);

## Description

This function sets the specified bit in reference memory. This function ensures overrides and transitions are taken into account for bit memory.

## InParam RefTable

Reference table segment selector used to indicate which table to access.

## InParam offset

Offset to use to clear the bit.

---

**Note:** This is 1-based. For example, use 1 to access %I00001.

---

## InParam bitNumber

For word type memories, this determines which bit to set. For bit type memories, this input is ignored. This is 1-based with a range of 1 to 16. For example, use 1 to set the least significant bit in a word memory.

## RetVal

The function returns GEF\_OK if the offset is within range or GEF\_ERROR if the offset is out of range. In the GEF\_ERROR case, the specified bit is not changed.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.



## clearBit

T\_INT32 clearBit(T\_WORD RefTable, T\_DWORD offset, T\_WORD bitNumber);

### Description

This function clears the specified bit in reference memory. This function ensures overrides and transitions are taken into account for bit memory.

### InParam RefTable

Reference table segment selector used to indicate which table to access.

### InParam offset

Offset to use to clear the bit.

---

**Note:** This is 1-based. For example, use 1 to access %I00001.

---

### InParam bitNumber

For word type memories, this determines which bit to clear. For bit type memories, this input is ignored. This is 1-based with a range of 1 to 16. For example, use 1 to clear the least significant bit in a word memory.

### RetVal

The function returns GEF\_OK if the offset is within range or GEF\_ERROR if the offset is out of range. In the GEF\_ERROR case, the specified bit is not changed.

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## rackX

T\_DWORD rackX(T\_BYTE rackNumber);

### Description

Returns the rack fault summary bit in the rack slot reference record based on the rackNumber. Only the first bit is significant. This indicates whether one or more modules in the rack are faulted.

### InParam rackNumber

Indicates which rack to get the fault summary bit from. rackNumber is 0 based and the maximum number of racks is specified in model\_specifics.h

## RetVal

Returns the rack fault summary bit for the requested rack in bit 0.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

## slotX

`T_DWORD slotX(T_BYTE rackNumber, T_DWORD slotNumber);`

## Description

Returns the fault bit for the specified slot and rack in the least significant bit.

## InParam rackNumber

Indicates which rack to use to get the fault bit. rackNumber is 0 based and the maximum number of racks is specified in `model_specifics.h`

## InParam slotNumber

Indicates which slot to use to get the fault bit. slotNumber is 0 based and the maximum number of racks is specified in `model_specifics.h`

## RetVal

Returns the fault bit for the requested rack and slot in bit 0.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

## blockX

`T_DWORD blockX(T_BYTE rackNumber, T_DWORD slotNumber,  
T_DWORD busNumber, T_DWORD sbaNumber);`

## Description

Returns the module fault reference bit for a particular block on the bus in the least significant bit.

## InParam rackNumber

Indicates which rack to use to get the module fault reference bit. rackNumber is 0 based and the maximum number of racks is specified in model\_specifics.h

## InParam slotNumber

Indicates which slot to use to get the module fault reference bit. slotNumber is 0 based and the maximum number of racks is specified in model\_specifics.h

## InParam busNumber

Indicates which bus to use to get the module fault reference bit. Valid values are 1 or 2.

## InParam sbaNumber

Indicates which serial bus offset to use to get the module fault reference bit.

sbaNumber is 0 based and the maximum number of modules per bus is specified in model\_specifics.h

## ReturnVal

Returns the module fault reference bit for the requested rack, slot, bus and serial bus address in bit 0.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are located in ctkPlcErrno.h.

## rsmb

```
RACK_REFERENCE_REC *rsmb(T_BYTE rackNumber);
```

```
typedef struct
```

```
{  
    T_DWORD RackFlags; /* Summary and failure flags */  
    T_DWORD SlotFaults; /* All 32 bits of Dword for slot fault bits */  
    T_DWORD BusRefs[MAX_NUM_BUSES_PER_SLOT]; /* Bus fault bits */  
    T_BYTE ModRefs[MAX_NUM_BUSES_PER_SLOT][LIMIT_NUM_SLOTS_PER_RACK]  
        [MAX_NUMBER_MODULES_PER_BUS/8];  
} RACK_REFERENCE_REC;
```

```
/* Definitions and Masks Used with RACK_REFERENCE_REC structure.
```

```
Note: LIMIT_NUM_SLOTS_PER_RACK & MAX_NUMBER_MODULES_PER_BUS  
are defined in model_specifics.h */
```

```
#define MAX_NUM_BUSES_PER_SLOT 2
#define REF_RACK_SUMMARY_FLAG 0x01
#define REF_RACK_FAILURE_FLAG 0x02
```

## Description

Returns a pointer to a `RACK_REFERENCE_REC` structure for the specified rack.

The `RACK_REFERENCE_REC` provides a structure to determine the location of faults anywhere in a single rack in the PLC system. The following notes provide details on how to use the structure when the return value is assigned to a pointer named `pRackRefRec`.

### Note:

<code>pRackRefRec-&gt;Flags</code>	If bit 0 is set, there is at least one module in the rack system with a fault. If bit 1 is set, the rack has a fault.
<code>pRackRefRec-&gt;Slot</code>	Each bit of this 32-bit variable represents 1 of 32 possible slots in the rack. If a bit is set (1), it indicates the module in the slot corresponding to the bit number (0–31) has a fault. For example, if <code>pRackRefRec-&gt;Slot</code> equals 0x0000000A, modules in slots 1 and 3 have faults because the 1st and 3rd bits are set.
<code>pRackRefRec-&gt;BusRefs[0]</code> (for bus 1) or <code>pRackRefRec-&gt;BusRefs[1]</code> (for bus 2)	Each bit of this 32-bit variable represents one of two possible busses on 1 of 32 possible slots in the rack. The bit is set if any modules on the bus have a fault. For example, if a Genius block on bus 1 has a fault for a GBC located in slot 3, the value of <code>pRackRef-&gt;BusRefs[0]</code> , assuming no other faulted modules, would be 0x00000008.
<code>pRackRefRec-&gt;ModRefs[BusNumber][SlotNumber][ModuleBytePosition]</code>	This gives an 8-bit variable where each bit represents whether a module on a bus connected to a rack based module in a slot has a fault. <code>BusNumber</code> can be a value of 0 or 1 for bus1 or bus2 respectively. <code>SlotNumber</code> can be a value from 0 to 31 representing the slot of the module that supports one of the busses. <code>ModuleBytePosition</code> is a value from 0 to 31 where each value represents 8 modules. For example, if <code>ModuleBytePosition</code> equals 0, it represents fault bits for modules at bus addresses 0 through 7. A value of 1 represents bus addresses 8 to 15, and so forth.  If a slot-based I/O modules does not have a bus associated with it, and if the modules has a fault, all <code>BusRefs</code> and <code>ModRefs</code> bits associated with that slot will be set.

## InParam rackNumber

Indicates which rack to use. `rackNumber` is 0 based and the maximum number of racks is specified in `model_specifics.h`

## RetVal

Returns pointer to a `RACK_REFERENCE_REC` structure.

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

### 3.5.11 Utility Function

The following utility function is described in `ctkPLCUtil.h`.

## PLCC\_Crc16Checksum

```
T_WORD PLCC_Crc16Checksum(T_BYTE *pFirstByte, T_DWORD length,  
                           T_WORD currentCrcValue);
```

### Description

This function calculates a CRC16 checksum over the given area with the given starting value and length in bytes. The `currentCrcValue` is normally 0. When checking a large memory range section by section, you can use the previous section's CRC value as the initial value.

### InParam pFirstByte

Pointer to the first byte to include in the checksum

### InParam length

Length of data in units of bytes to calculate the checksum

### InParam currentCrcValue

The initial CRC value from the previous CRC calculation when creating CRC over multiple sections.

### ReturnVal

Returns the CRC16 checksum.

## Errno

This function sets Errno if `pFirstByte` is a null pointer. See `cpuErrno.h` for possible values.

## 3.5.12 Errno Functions

Some functions provide status by setting a global errno variable. To effectively examine the value of errno, you should:

1. Call `PLCC_ClearErrno` to make sure errno was not set by a previous function call.
2. Call the desired function that can potentially set errno.
3. Call `PLCC_GetErrno` to get the current errno value.

Any non-zero errno value indicates an error. The errno definitions are described in

`cpuErrno.h`. (`\Targets\CommonFiles\IncCommon\PlcInc`) and

`Errno.h` (`\Targets\CommonFiles\IncCommon\VxCommon`).

The PLCC Errno Functions are described in `ctkPlcErrno.h`.

### PLCC\_GetErrno

`int PLCC_GetErrno(void);`

#### Description

This function returns the errno value in the current context. The errno value is an error code set by the last PLC Target Library or C Run Time Library function to declare an error.

#### RetVal

Returns the errno value.

### PLCC\_ClearErrno

`void PLCC_ClearErrno(void);`

#### Description

This function sets the Errno value in the current context to 0.

## 3.5.13 PLC Variable Access

The C toolkit can access PLC variables, which are declared on the PLC and can be managed variables, I/O variables, or mapped variables. This section describes the macros and external functions (externs) used for accessing PLC variables. These macros and functions are described in `ctkVariables.h`

---

**Note:**

- When reading/writing non-array variables or individual elements of arrays for user data types, coherency will be guaranteed for the entire read or write.
  - For string variables, the data is not guaranteed to be coherent.
  - When reading/writing non-boolean array variables, coherency will be guaranteed for each individual element of the array.
  - This feature is supported only on versions 3.50 and later.
- 

## Type and Structure Definitions

### PLC\_VAR

```
#define PLC_VAR(VariableRecord, PlcVariableName)  
PLC_VAR_ENTRY_RECORD(VariableRecord, PlcVariableName)
```

#### Description

This macro is used to create a reference to a PLC variable in C logic. These should be declared as variables global to the C applications. All variables used in C applications must be internally or externally published in the PLC.

#### InParam VariableRecord

Name for a reference variable of type `PLC_VAR_REC` that will be used to reference the PLC variable when calling routines in this module. This input parameter must be a valid "C" variable name.

#### InParam PlcVariableName

Exact name of the PLC variable to be accessed within quotes (for example, "myPlcVar").

#### Example 1

For a PLC variable named `motorPosition`:

```
PLC_VAR(motorPositionRec, "motorPosition");
```

To use this in multiple C files for a single application, place the following extern statement in a header file:

```
extern PLC_VAR_REC motorPositionRec;
```

#### Example 2

For a 3 x 5 array of WORDs named `algDiagnostics`:

```
PLC_VAR(algDiagnosticRec, "algDiagnostics");
```

When calling ReadPlcVar and WritePlcVar with this declaration, the entire 3 x 5 array is read/written. ReadPlcArrayVarElement and WritePlcArrayVarElement can be used to access individual elements of the array.

### Example 3

For a 3 x 5 array of WORDs named algDiagnostics where access to a single element is needed:

```
PLC_VAR(algDiagnosticElemRec, "algDiagnostics[2,1]");
```

When calling ReadPlcVar and WritePlcVar with this declaration, a single word is read/written. ReadPlcArrayVarElement and WritePlcArrayVarElement called with this declaration would return an error.

### Example 4

For an array of custom structures named mainValves with a member flowRate:

```
PLC_VAR(mainValveFlowRateRec, "mainValves[3,4].flowRate");
```

Members of structures must be accessed independently. Declaring a PLC\_VAR with only "mainValves" or "mainValves[3,4]" will result in an error when attempting to store logic.

## PLC Var 'C' Types

```
typedef T_BYTE          PLC_VAR_BYTE;
typedef T_WORD          PLC_VAR_WORD;
typedef T_INT16         PLC_VAR_INT;
typedef T_UINT16        PLC_VAR_UINT;
typedef T_DWORD         PLC_VAR_DWORD;
typedef T_INT32         PLC_VAR_DINT;
typedef float          PLC_VAR_REAL;
typedef T_BOOLEAN      PLC_VAR_BOOL;      /* This should be used for a single BOOL
variable only. PLC_VAR_BYTE should be used for arrays of BOOLs because the bits are
packed into bytes. */
```

## Routines

### Proc ReadPlcVar

```
extern T_INT32 ReadPlcVar(PLC_VAR_REC *pVarInfo, void *pReadTo);
```

#### Description

This function reads the value of a PLC variable into a buffer provided by the caller.

#### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the variable to be read.

#### InParam pReadTo



Pointer to the memory location where the value of the variable to be read should be located.

---

**Note:** If *pVarInfo* references an array, the entire array will be read.

- For type *BOOL*, an entire byte will be written to *pReadTo* with the low bit of the byte containing the value of the *BOOL* variable. The remaining seven bits are zero filled.
  - For an array of type *BOOL*, the number of bytes written will be  $(\text{total number of elements} + 7) / 8$ . The first bit will be written to the least significant bit of the first byte. The data written will be byte aligned even if the PLC variable is not. Bits that are not part of the array are zero filled.
  - For type *BYTE*, an 8-bit value will be written to *pReadTo* (*BYTES* mapped to non-discrete memories, such as *%R* or *%W*, consume 16 bits on the PLC, but will be packed when written to *pReadTo* by this routine).
  - For type *STRING*, the size in bytes of the data written to *pReadTo* will be the "max length" in the variable's declaration on the PLC.
- 

## ReturnVal

GEF\_OK if successful.

GEF\_ERROR for bad parameter. (Use *Errno* to determine cause.)

## Errno

If there is an error, this function sets *Errno* to give specific information on what caused the error. Applications that use *Errno* should first call *PLCC\_ClearErrno* to ensure *Errno* was not already set by another function call. *Errno* can be read using *PLCC\_GetErrno*. *Errno* values are located in *ctkPlcErrno.h*.

## Proc ReadPlcArrayVarElement

```
extern T_INT32 ReadPlcArrayVarElement(PLC_VAR_REC *pVarInfo,  
                                     void *pReadTo,  
                                     T_INT32 numIndices,  
                                     ...);
```

## Description

This function reads the value of a single element in a PLC array variable into a buffer provided by the caller.

### InParam pVarInfo

Pointer to a *PLC\_VAR\_REC* information record for the array containing the element to be read.

### InParam pReadTo

Pointer to the memory location where the value of the variable to be read is located.

---

**Note:**

- For type *BOOL*, an entire byte will be written to *pReadTo* with the low bit of the byte containing the value of the *BOOL* variable.
  - For type *BYTE*, an 8-bit value will be written to *pReadTo* regardless of whether the array is in discrete or non-discrete memory on the PLC (*BYTES* mapped to non-discrete memories, such as *%R* or *%W*, consume 16 bits on the PLC).
  - For type *STRING*, the size in bytes of the data written to *pReadTo* will be the "max length" in the variable's declaration on the PLC.
- 

### InParam numIndices

Number of indices needed to locate an element of the array. This must be greater than zero and must match the number of dimensions of the variable declared on the PLC.

### InParam <indices>

A variable number of indices (must match numIndices) indicating the element of the array to be read. These should be *T\_INT32* type.

### RetVal

*GEF\_OK* if successful.

*GEF\_ERROR* for bad parameter. (Use *Errno* to determine cause.)

### Errno

If there is an error, this function sets *Errno* to give specific information on what caused the error. Applications that use *Errno* should first call *PLCC\_ClearErrno* to ensure *Errno* was not already set by another function call. *Errno* can be read using *PLCC\_GetErrno*. *Errno* values are provided in *ctkPlcErrno.h*.

## Proc ReadPlcVarDiag

```
extern T_INT32 ReadPlcVarDiag(PLC_VAR_REC *pVarInfo, void *pReadDiagsTo);
```

### Description

This function reads the diagnostic value(s) for a PLC variable into a buffer provided by the caller. If the variable does not have diagnostics, an error will be returned. Variables of type STRING are invalid and will return an error.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the variable to be read.

### InParam pReadDiagsTo

Pointer to the memory location where the diagnostic values of the variable should be written.

---

#### Note:

- If pVarInfo references an array, the diagnostics for the entire array will be read.
  - For type BOOL, an entire byte will be written to pReadDiagsTo with the low bit of the byte containing the diagnostic value of the BOOL variable.
  - For an array of type BOOL, the number of bytes written will be (total number of elements + 7) / 8. The first diagnostic bit will be written to the least significant byte of the first byte.
  - For type BYTE and BYTE arrays, there will be one byte of diagnostic written for every byte element.
  - For all other types, the number of bytes written will be the byte size of the PLC variable divided by 2 if the variable is in non-discrete memory. For example, an array of 8 words would have 8 bytes of diagnostic data. If the variable is discrete memory, the number of bytes written will be equal to the size of the array variable in bytes.
- 

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are provided in ctkPlcErrno.h.

### Bit Masks to be Used with Diagnostics

These bit masks are defined in ctkRefMem.h.

For access to analog input diagnostic memory:

HI_ALARM_MSK	0x02
LO_ALARM_MSK	0x01
AI_OVERRANGE_MSK	0x08
AI_UNDERRANGE_MSK	0x04

For access to analog output diagnostic memory:

AQ_OVERRANGE_MSK	0x40
AQ_UNDERRANGE_MSK	0x20

## Proc ReadPlcArrayVarElementDiag

```
extern T_INT32 ReadPlcArrayVarElementDiag(PLC_VAR_REC *pVarInfo,  
                                           void *pReadDiagsTo,  
                                           T_INT32 numIndices,  
                                           ...);
```

### Description

This function reads the diagnostic values for a single element in a PLC array variable into a buffer provided by the caller.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the array containing the element whose diagnostics are to be read. If the variable does not have diagnostics, an error will be returned. Arrays of STRINGs are invalid and will return an error.

### InParam pReadDiagsTo

Pointer to the memory location where the diagnostic values are to be written.

---

#### Note:

- For type *BOOL*, an entire byte will be written to *pReadDiagsTo* with the low bit of the byte containing the diagnostic value of the *BOOL* variable.
  - For type *BYTE*, 8 bits of diagnostics will be written to *pReadTo* regardless of whether the array is in discrete or non-discrete memory on the PLC.
  - For all other types, if the variable is in non-discrete memory, the number of bytes written will be the byte size of the array element divided by 2.
- 

For example, an element from an array of words would be 1 byte of diagnostic data. If the variable is discrete memory, the number of bytes written will be equal to the size of an array element in bytes.

### InParam numIndices

Number of indices needed to locate an element of the array. This must be greater than zero and must match the number of dimensions of the variable declared on the PLC.

### InParam <indices>

A variable number of indices (must match *numIndices*) that indicates the element of the array for which diagnostics is to be read. These should be *T\_INT32* type.

### Errno

If there is an error, this function sets *Errno* to give specific information on what caused the error. Applications that use *Errno* should first call *PLCC\_ClearErrno* to ensure *Errno* was not already set by another function call. *Errno* can be read using *PLCC\_GetErrno*. *Errno* values are provided in *ctkPlcErrno.h*.

## Bit Masks to be Used with Diagnostics

These bit masks are defined in ctkRefMem.h.

For access to analog input diagnostic memory:

HI_ALARM_MSK	0x02
LO_ALARM_MSK	0x01
AI_OVERRANGE_MSK	0x08
AI_UNDERRANGE_MSK	0x04

For access to analog output diagnostic memory:

AQ_OVERRANGE_MSK	0x40
AQ_UNDERRANGE_MSK	0x20

## Proc ReadPlcVarOvr

```
extern T_INT32 ReadPlcVarOvr(PLC_VAR_REC *pVarInfo, void *pReadOvrTo);
```

### Description

This function reads the override value(s) for a PLC variable into a buffer provided by the caller.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the variable to be read. If the variable does not have overrides, an error will be returned.

### InParam pReadOvrTo

Pointer to the memory location where the override values for the variable should be written.

---

**Note:** If *pVarInfo* references an array, the overrides for the entire array will be read.

- For type *BOOL*, an entire byte will be written to *pReadOvrTo* with the low bit of the byte containing the override value for the *BOOL* variable.
  - For an array of type *BOOL*, the number of bytes written will be  $(\text{total number of elements} + 7) / 8$ . The first override bit will be written to the least significant byte of the first byte.
  - For all other types, the number of bytes written will be equal to the byte size of the PLC variable.
- 

## Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are provided in ctkPlcErrno.h.

## Proc ReadPlcArrayVarElementOvr

```
extern T_INT32 ReadPlcArrayVarElementOvr(PLC_VAR_REC *pVarInfo,  
                                          void *pReadOvrTo,  
                                          T_INT32 numIndices,  
                                          ...);
```

### Description

This function reads the override value(s) for a single element in a PLC array variable into a buffer provided by the caller.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the array containing the element whose diagnostics are to be read. If the variable does not have overrides, an error will be returned.

### InParam pReadOvrTo

Pointer to the memory location where the override values are to be written.

---

#### Note:

- For type *BOOL*, an entire byte will be written to *pReadOvrTo* with the low bit of the byte containing the override value for the *BOOL* variable.
  - For all other types, the number of bytes written will be equal to the byte size of an element in the PLC array variable.
- 

### InParam numIndices

Number of indices needed to locate an element of the array. This must be greater than zero and must match the number of dimensions of the variable declared on the PLC.

### InParam <indices>

A variable number of indices (must match *numIndices*) indicating the element of the array for which diagnostics are to be read. These should be *T\_INT32* type.

### Errno

If there is an error, this function sets *Errno* to give specific information on what caused the error. Applications that use *Errno* should first call *PLCC\_ClearErrno* to ensure *Errno* was not already set by another function call. *Errno* can be read using *PLCC\_GetErrno*. *Errno* values are provided in *ctkPlcErrno.h*.

## Proc ReadPlcVarTrans

```
extern T_INT32 ReadPlcVarTrans(PLC_VAR_REC *pVarInfo, void *pReadTransTo);
```

### Description

This function reads the transition value(s) for a PLC variable into a buffer provided by the caller.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the variable to be read. If the variable does not have transitions, an error will be returned.

### InParam pReadTransTo

Pointer to the memory location where the transition values for the variable should be written.

---

#### Note:

- If *pVarInfo* references an array, the transitions for the entire array will be read.
  - For type *BOOL*, an entire byte will be written to *pReadTransTo* with the low bit of the byte containing the transition value for the *BOOL* variable.
  - For an array of type *BOOL*, the number of bytes written will be  
$$(total\ number\ of\ elements + 7) / 8.$$

The first transition bit will be written to the least significant byte of the first byte.
  - For all other types, the number of bytes written will be equal to the byte size of the PLC variable.
- 

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are located in `ctkPlcErrno.h`.

## Proc ReadPlcArrayVarElementTrans

```
extern T_INT32 ReadPlcArrayVarElementTrans(PLC_VAR_REC *pVarInfo,  
                                           void *pReadTransTo,  
                                           T_INT32 numIndices,  
                                           ...);
```

### Description

This function reads the transition value(s) for a single element in a PLC array variable into a buffer provided by the caller.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the array containing the element whose diagnostics are to be read. If the variable does not have transitions, an error will be returned.

### InParam pReadTransTo

Pointer to the memory location where the transition values are to be written.

---

#### Note:

- For type *BOOL*, an entire byte will be written to *pReadTransTo* with the low bit of the byte containing the transition value for the *BOOL* variable.
  - For all other types, the number of bytes written will be equal to the byte size of an element in the PLC array variable.
- 

### InParam numIndices

Number of indices needed to locate an element of the array. This must be greater than zero and must match the number of dimensions of the variable declared on the PLC.

### InParam <indices>

A variable number of indices (must match *numIndices*) indicating the element of the array for which diagnostics are to be read. These should be *T\_INT32* type.

### Errno

If there is an error, this function sets *Errno* to give specific information on what caused the error. Applications that use *Errno* should first call *PLCC\_ClearErrno* to ensure *Errno* was not already set by another function call. *Errno* can be read using *PLCC\_GetErrno*. *Errno* values are provided in *ctkPlcErrno.h*.



## Proc WritePlcVar

extern T\_INT32 WritePlcVar(PLC\_VAR\_REC \*pVarInfo, void \*pWriteFrom);

### Description

This function writes a value to a PLC variable from the buffer provided by the caller. This routine accounts for overrides and transitions when applicable.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the variable to be written.

### InParam pWriteFrom

Pointer to the memory location of the value(s) to be written to the PLC variable.

---

#### Note:

- If *pVarInfo* references an array, the entire array will be written.
  - For type *BOOL*, the least significant bit at the byte pointed to by *pWriteFrom* will be written to the PLC variable.
  - For an array of type *BOOL*, the bits will be copied starting at the least significant bit of the byte pointed to by *pWriteFrom*.
  - For type *BYTE*, an 8-bit value will be read from *pWriteFrom* (For non-discrete memories where the *BYTE* variable consumes 16 bits on the PLC the 8-bit value will be written to the least significant 8 bits of the 16-bit word).
  - For type *STRING*, the size of the data copied from *pWriteFrom* will be the "max length" in the variable's declaration.
- 

### RetVal

GEF\_OK if successful.

GEF\_ERROR for bad parameter. (Use Errno to determine cause.)

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call `PLCC_ClearErrno` to ensure Errno was not already set by another function call. Errno can be read using `PLCC_GetErrno`. Errno values are provided in `ctkPlcErrno.h`.

## Proc WritePlcArrayVarElement

```
extern T_INT32 WritePlcArrayVarElement(PLC_VAR_REC *pVarInfo,  
                                       void *pWriteFrom,  
                                       T_INT32 numIndices,  
                                       ...);
```

### Description

This function writes a single element in a PLC array variable from a buffer provided by the caller.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record for the array containing the element to be written.

### InParam pWriteFrom

Pointer to the memory location containing the value to be written to the array element.

---

#### Note:

- For type *BOOL*, the least significant bit at the byte pointed to by *pWriteFrom* will be written to the PLC variable array element.
  - For type *BYTE*, an 8-bit value will be read from *pWriteFrom* (For non-discrete memories where the *BYTE* variable consumes 16 bits on the PLC the 8-bit value will be written to the least significant 8 bits of the 16-bit word).
  - For type *STRING*, the size of the data copied from *pWriteFrom* will be the "max length" in the variable's declaration.
- 

### InParam numIndices

Number of indices needed to locate an element of the array. This must be greater than zero and must match the number of dimensions of the variable declared on the PLC.

### InParam <indices>

A variable number of indices (must match *numIndices*) indicating the element of the array to be written. These should be *T\_INT32* type.

### ReturnVal

GEF\_OK if successful.

GEF\_ERROR for bad parameter. (Use *Errno* to determine cause.)

### Errno

If there is an error, this function sets *Errno* to give specific information on what caused the error. Applications that use *Errno* should first call *PLCC\_ClearErrno* to ensure *Errno* was not already set by another function call. *Errno* can be read using *PLCC\_GetErrno*. *Errno* values are located in *ctkPlcErrno.h*.

## Proc PlcVarMemCopy

```
extern T_INT32 PlcVarMemCopy(PLC_VAR_REC *pDestVarInfo,  
                             PLC_VAR_REC *pSrcVarInfo);
```

### Description

This function copies the contents of one PLC variable to another PLC variable of the same type and size. The size of the destination variable must be greater than or equal to the size of the source variable. No other type or bounds checking will be done.

### InParam pDestVarInfo

Pointer to a PLC\_VAR\_REC information record for the destination variable.

### InParam pSrcVarInfo

Pointer to a PLC\_VAR\_REC information record for the destination variable.

### RetVal

GEF\_OK if successful.

GEF\_ERROR for bad parameter. (Use Errno to determine cause.)

### Errno

If there is an error, this function sets Errno to give specific information on what caused the error. Applications that use Errno should first call PLCC\_ClearErrno to ensure Errno was not already set by another function call. Errno can be read using PLCC\_GetErrno. Errno values are provided in ctkPlcErrno.h.

## Proc PlcVarType

typedef enum

```
{
    PLC_BOOL_VAR_TYPE      = 0,
    PLC_BYTE_VAR_TYPE      = 13,
    PLC_WORD_VAR_TYPE      = 14,
    PLC_INT_VAR_TYPE       = 25,
    PLC_UINT_VAR_TYPE      = 26,
    PLC_DWORD_VAR_TYPE     = 18,
    PLC_DINT_VAR_TYPE      = 1,
    PLC_REAL_VAR_TYPE      = 27,
    PLC_STRING_VAR_TYPE    = 24,
    PLC_INVALID_VAR_TYPE   = 0xFFFFFFFF
} PLC_VAR_TYPES;
```

extern T\_DWORD PlcVarType(PLC\_VAR\_REC \*pVarInfo);

### Description

This function returns the type value for a PLC variable.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### ReturnVal varType

Value defining the type of the PLC variable. Returns PLC\_INVALID\_VAR\_TYPE if input is NULL.

## Proc PlcVarSizeof

```
extern T_DWORD PlcVarSizeof(PLC_VAR_REC *pVarInfo);
```

### Description

This function returns the total size of a PLC variable. If the variable is a BOOL or array of BOOLS, the size is in bits. For all other types, the size is in bytes.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### RetVal size

Size in bits for BOOLS/Arrays of BOOLS. Size in bytes for all other types. Zero is returned for NULL input pointer.

---

**Note:** *BYTE arrays in non-discrete memory are not packed on the PLC, so each byte occupies 16 bits of PLC memory. This routine will return the size in bytes as if the byte array were packed, not the size of the memory occupied on the PLC.*

---

## Proc PlcVarSizeofDiag

```
extern T_DWORD PlcVarSizeofDiag(PLC_VAR_REC *pVarInfo);
```

### Description

This function returns the total size of the diagnostic memory for a PLC variable. If the variable is a BOOL or array of BOOLS, the size is in bits. For all other types, the size is in bytes.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### RetVal size

Size in bits for BOOLS/Arrays of BOOLS. Size in bytes for all other types. Zero is returned for NULL input pointer.

---

**Note:** *BYTE arrays in non-discrete memory are not packed on the PLC, therefore each byte occupies 16 bits of PLC memory. This routine will return the size in bytes as if the byte array were packed, not the size of the memory occupied on the PLC.*

---

## Proc PlcVarSizeofOvr

```
extern T_DWORD PlcVarSizeofOvr(PLC_VAR_REC *pVarInfo);
```

### Description

This function returns the total size of the override memory for a PLC variable. If the variable is a BOOL or array of BOOLS, the size is in bits. For all other types, the size is in bytes.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### ReturnVal size

Size in bits for BOOLS/Arrays of BOOLS. Size in bytes for all other types. Zero is returned for NULL input pointer.

---

**Note:** *BYTE arrays in non-discrete memory are not packed on the PLC, so each byte occupies 16 bits of PLC memory. This routine returns the size in bytes as if the byte array were packed, not the size of the memory occupied on the PLC.*

---

## Proc PlcVarSizeofTrans

```
extern T_DWORD PlcVarSizeofTrans(PLC_VAR_REC *pVarInfo);
```

### Description

This function returns the total size of the transition memory for a PLC variable. If the variable is a BOOL or array of BOOLS, the size is in bits. For all other types, the size is in bytes.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### ReturnVal size

This function returns size in:

Bits for BOOLS/Arrays of BOOLS

Bytes for all other types.

Zero is returned for NULL input pointer.

---

**Note:** *BYTE arrays in non-discrete memory are not packed on the PLC, so each byte occupies 16 bits of PLC memory. This routine will return the size in bytes as if the byte array were packed, not the size of the memory occupied on the PLC.*

---

## Proc PlcVarNumDimensions

```
extern T_DWORD PlcVarNumDimensions(PLC_VAR_REC *pVarInfo);
```

### Description

This function returns the number of dimensions for a PLC variable. If the variable is not an array, zero is returned.

---

**Note:** A variable of type *STRING* will return zero. An array of *STRING*s will return non-zero.

---

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### RetVal numDimensions

Number of dimensions for array variables, zero for scalar types.

## Proc PlcVarHasDiags

```
extern T_BOOLEAN PlcVarHasDiags(PLC_VAR_REC *pVarInfo);
```

### Description

This function returns TRUE if the PLC variable supports diagnostics, FALSE if not.

---

**Note:** This routine returns TRUE if the variable supports diagnostics regardless of the state of the diagnostic data.

---

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### RetVal

TRUE if the PLC variable has diagnostic values associated with it, FALSE if not.

## Proc PlcVarHasOverrides

```
extern T_BOOLEAN PlcVarHasOverrides(PLC_VAR_REC *pVarInfo);
```

### Description

This function returns TRUE if the PLC variable supports overrides, FALSE if not.

---

**Note:** This routine returns TRUE if the variable supports overrides regardless of the state of the overrides.

---

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### RetVal

TRUE if the PLC variable has override values associated with it, FALSE if not.

## Proc PlcVarHasTransitions

**extern T\_BOOLEAN PlcVarHasTransitions(PLC\_VAR\_REC \*pVarInfo);**

### Description

This function returns TRUE if the PLC variable supports transitions, FALSE if not.

---

**Note:** *This routine returns TRUE if the variable supports transitions regardless of the state of the transitions.*

---

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### RetVal

TRUE if the PLC variable has transition values associated with it, FALSE if not.

## Proc PlcVarArrayElementSize

**extern T\_DWORD PlcVarArrayElementSize(PLC\_VAR\_REC \*pVarInfo);**

### Description

This function returns the size in bytes of an individual element of an array variable. If the variable is a BOOL, an array of BOOLs, or not an array, zero will be returned.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### RetVal

Size in bytes of an individual array element.

## Proc PlcVarArrayBound

**extern T\_DWORD PlcVarArrayBound(PLC\_VAR\_REC \*pVarInfo, T\_DWORD dimension);**

### Description

This function returns the upper boundary for a given array dimension. For example, if the variable is a 3 by 5 array, requesting dimension 1 would return 3 and requesting dimension 2 would return 5. If the variable is not an array or the variable does not have as many dimensions as indicated by the "dimension" input parameter, zero is returned.

### InParam pVarInfo

Pointer to a PLC\_VAR\_REC information record.

### InParam dimension

Indicates the array dimension to return the bound for.

### RetVal

Boundary of the requested array dimension.



## 3.6 Application Considerations

When creating a C application, it is necessary to keep in mind a few items regarding the target PACSystems:

1. How big is each of the target PLC's reference memories?
2. Will the block be called from the MAIN ladder block or from some other ladder block?
3. How large is the C application likely to be?

All of these questions must be kept in mind while developing C applications. The following sections provide detail on each of these questions and other questions regarding the creation of C applications.

### 3.6.1 Application File Names

Application file names are limited to 31 characters. The first character in the filename must be alphabetic.

### 3.6.2 Floating Point Arithmetic

All PACSystems CPUs support floating point math.

### 3.6.3 Available Reference Data Ranges

When a C application uses an index variable to select an element from PLC reference memory via a reference memory macro, the value of the index variable should always be checked against the size of the target PLC's reference memory. It is also a good practice to check the size before calling reference memory functions but is not absolutely necessary because the function will return an error status or set Errno if the index variable is out of range for the selected memory. The size of any PLC reference memory can be determined using the corresponding SIZE macro. As an example, consider the following ladder logic rung and sample block, where the value in %P1 is to be used as an index into %R memory and the value at %R[%P1] is to be copied to %P2:

Figure 13:



## Range Checking Indirect References Using the SIZE Macros

```
/* The value at x1 will be used as an index into */
/* register memory. The value at %R(x1) will be */
/* copied to y1. */

int GefMain(T_WORD *x1, T_INT16 *y1)
{
    /* FIRST - check X1 & Y1 != NULL */
    /* SECOND - must range check value at x1 to ensure */
    /* that we will stay within limits of PLC */
    /* %R reference memory. */
    if ((x1 != NULL) && y1 != NULL) {
        if (*x1 > R_SIZE) return(ERROR);

        /* Range check proved OK ==> go ahead and copy data */
        *y1 = RW(*x1);
        return(GEF_EXECUTION_OK);
    }
    else return (GEF_EXECUTION_ERROR);
}
```

In the above example, the index `*x1` is compared to `R_SIZE`. If the target PLC contains 1024 registers, then `R_SIZE` will evaluate to 1024. If `*x1` is greater than 1024 (`R_SIZE`), the program will return with the status `GEF_EXECUTION_ERROR` which indicates that the ENO output of the CALL function block should be turned OFF. With `*x1` greater than `R_SIZE`, the C block will return with `GEF_EXECUTION_ERROR` status and no attempt is made to index into register memory nor to copy any register memory value to `*y1`.

## 3.6.4 Global Variable Initialization

Global variables can be used by C applications running in a PACSystems control system. Global variables are those which are declared outside of a function, typically outside of and before **GefMain()**. Both initialized and uninitialized global variables may be used.

```
T_INT32 xyz; /* uninitialized global var */
T_INT32 abc = 123; /* initialized global var */

int GefMain() {
    xyz = RW(1);
    RI(2) = ++abc;
    return(GEF_EXECUTION_OK);
}
```

When a C application is compiled and linked to form relocate-able (**.gefElf**) file, all global variables have a relative location within the **.gefElf** image. If the global variable is declared in the C source to have an initial value, the location in the **.gefElf** image for that global variable will contain the initialized value. When a C application is incorporated into a Machine Edition folder and that folder is stored to a PACSystems CPU, the CPU stores an image of the **.gefElf** file into user memory with space pre-allocated for all global variables and with all initialized global variables already containing their predefined values. Upon storing the **.gefElf** image, the PLC will make a copy of the data portion (data portion = initialized global variables).

Once the PLC is placed into **RUN** mode, the C application may operate upon any of its global variables. Each of the C application's global variables will retain its value from one sweep to the next sweep and will continue to do so until the PLC goes to **STOP** mode. On the transition from **STOP** mode to **RUN** mode, the PLC will re-initialize all of the C application's initialized global data to those values in the saved copy of global data start values. (Recall that the start values were saved when the folder was stored to the PLC.)

## 3.6.5 Static Variables

The keyword “static” may be used with either global variables or variables declared inside a function (including **GefMain()**). These variables will retain their value from sweep to sweep like global data. If a static variable is declared with an initial value, the variable will be initialized on the first execution from store or on transition from **STOP** to **RUN** mode. If a static variable is declared without an initial value, the initial value is undefined and **must** be initialized by the C application.

---

**Note:** *If C blocks are used multiple times in a ladder, static or global variables may not contain expected data from sweep to sweep. Multiple use blocks must at least receive a unique ID for each call or a unique work area to properly distinguish multiple calls.*

---

## 3.6.6 Data Retentiveness

All global variables and static variables are either retentive or non-retentive. Values of retentive data are preserved across both power-cycles (assuming a good battery is attached) and stop-to-run transitions. Non-retentive data is reinitialized on each stop-to-run transitions using values saved when the application was first stored.

All global and static variables, which are given an initial value, will be non-retentive. In general, uninitialized global data will be retentive. Since non-retentive data requires twice the memory space within the CPU (one for the working copy, and one for the saved copy), large initialized data structures should be avoided if memory usage is a concern.

The following examples illustrate retentive and non-retentive variables.

### Examples:

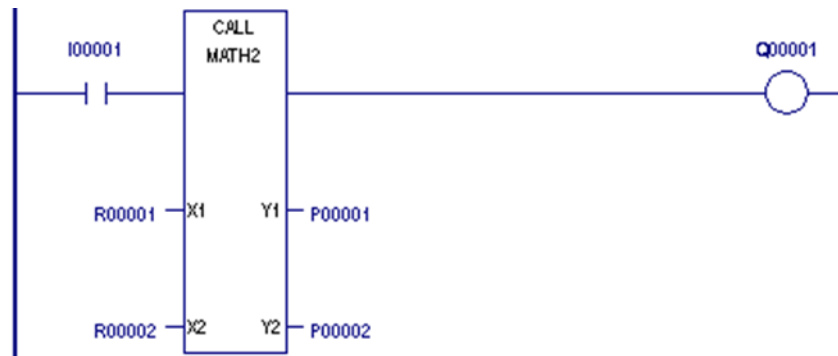
```
T_INT16 my_var1;           /* retentive */
T_INT16 my_var2 = 20;      /* non-retentive; reset to 20 on stop-to-run transitions */
static T_INT16 my_var3;    /* retentive */
static T_INT16 my_var4 = 12; /* non-retentive, reset to 12 on stop-to-run transitions */
```

## 3.6.7 GefMain( ) Parameter Declaration Errors for Blocks

When declaring the parameters to GefMain( ) in a block, the type, order, and number of parameters must match the ladder logic call instruction exactly. Use the following ladder logic segment and associated C block as an example:

```
/* This rung of ladder logic calls MATH2 to      */
/* add the two integers X1 and X2 and place the sum in Y1 */
/* and subtract the integer X2 from the integer X1, placing */
/* the difference in Y2.                               */
```

Figure 14:



```
/* MATH2 :
 * This function has two input parameters and two output
 * parameters.
 * Y1 = X1 + X2;
 * Y2 = X1 - X2;
 */

int GefMain( T_INT16 *x1, T_INT16 *x2, T_INT16 *y1, T_INT16 *y2){
    if (((x1 != NULL) && (y1 != NULL)) &&
        ((x2 != NULL) && (y2 != NULL))) {
        *y1 = *x1 + *x2;
        *y2 = *x1 - *x2;
        return(GEF_EXECUTION_OK);
    }
    else return (GEF_EXECUTION_ERROR);
}
```

As written above, the example is correct; the ladder logic call and the block declaration match. The operation of the ladder logic and the block will execute properly.

## Type Mismatch Errors

If, however, the block declaration is changed to the following, execution errors will occur.

```
int GefMain( T_REAL32 *x1, T_REAL32 *x2, T_REAL32 *y1, T_REAL32 *y2) {
    if (((x1 != NULL) && (y1 != NULL)) &&
        ((x2 != NULL) && (y2 != NULL))) {
        *y1 = *x1 + *x2;
        *y2 = *x1 - *x2;
        return(GEF_EXECUTION_OK);
    }
    else return (GEF_EXECUTION_ERROR);
}
```

The block will compile and link without error. The .gefElf file will be added and imported to the application folder without error. Similarly, the folder will store to the PACSystems CPU without error. No error will appear until the ladder and block are executed. The ladder logic will call MATH2 passing pointers to two (2) input parameters and pointers to two (2) output parameters. MATH2 expects two (2) input parameter pointers and two (2) output parameter pointers. The error occurs because the ladder logic uses integer variables (16 bits each), but the block uses float variables (32 bits each). This results in the block using the pointer x1 to read a 32-bit floating point value which starts at %R1 (the value used in the ladder logic). The 32-bit floating point value starting at %R1 includes both %R1 and %R2, but %R2 is the reference specified in ladder logic as x2. Since the input variables overlap, unpredictable values will result from the execution of this block. Notice also that the output parameters will have a similar problem.

## Parameter Ordering Errors

Execution errors can also occur due to differences in the order of the parameters when calling a block and the order of the parameters in the block declaration of **GefMain()**. Continuing with the same example, if the ladder logic is unchanged but **GefMain()** is declared as follows, an execution error will occur.

```
int GefMain ( T_INT16 *x1, T_INT16 *y1, T_INT16 *x2, T_INT16 *y2) {
    ...
}
```

No error message will be generated, just unpredictable output values. The execution error occurs because ladder logic always passes all of the specified input parameters in top-to-bottom order, followed by all of the specified output parameters, also in top-to-bottom order. In this case, the ladder logic passes %R1, %R2, %P1, and %P2, the two input parameters followed by the two output parameters. The block associates the parameters from the ladder logic call with its own variable names, as in the following example:

```
T_INT16 *x1 refers to %R1
T_INT16 *y1 refers to %R2
T_INT16 *x2 refers to %P1
T_INT16 *y2 refers to %P2
```

When the block executes the statement:

```
*y1 = *x1 + *x2;
```

the resulting operation adds the contents of %R1 (\*x1) to the contents of %P1 (\*x2) and place the sum in %R2 (\*y1), which is not what the ladder logic program expects.

Since the ladder logic call to a block always specifies the parameters in order (inputs 1 to 63) and (outputs 1 to 63), the block declaration of **GefMain()** must specify the parameters to **GefMain()** in the same order.

## Parameter Number Errors

If the number of parameters associated with a block in ladder logic does not match the number of parameters in the declaration of **GefMain()** for the block, potentially severe execution errors will occur.

---

**Note:** *It is essential that the number of parameters in a call to a block and the actual number of parameters required by the called block match; otherwise, the block will use invalid pointer variables to perform reads and writes.*

---

Again, using our example with the ladder logic portion unchanged, the effect of a difference in the number of parameters can be illustrated in the following example:

```
int GefMain ( T_INT16 *x1, T_INT16 *y1) {
/* Add the contents of %R1 to the contents pointed to by x1 */
/* and then store the sum in the location pointed to by y1 */
    if ((x1 != NULL) && (y1 != NULL)) {
        *y1 = *x1 + RI(1);
        return(GEF_EXECUTION_OK);
    }
    else return (GEF_EXECUTION_ERROR);
}
```

In this scenario, the ladder logic call will pass four parameters, %R1, %R2, %P1, and %P2. The block expects two parameters, x1 and y1, which it will associate with the passed in parameters as follows:

```
T_INT16 *x1 refers to %R1
T_INT16 *y1 refers to %R2
%P1 and %P2 are not referenced
```

The operation of this block with regard to parameter x1 is flawless. However, when y1 is used as the pointer for storing the sum, the sum will be written to %R2, not to %P1. This will cause incorrect operation of the application.

A more severe scenario is a block declared as follows:

```
int GefMain (T_INT16 *x1, T_INT16 *x2, T_INT16 *x3, T_INT16 *y1, T_INT16 *y2, T_INT16*y3) {  
/* Add the contents of %Rn to the contents pointed to by xn */  
/* and then store the sum in the location pointed to by yn */  
    *y1 = *x1 + RI(1);  
    *y2 = *x2 + RI(2);  
    *y3 = *x3 + RI(3);  
    return(GEF_EXECUTION_OK);  
}
```

The above block can have catastrophic results if executed in conjunction with the example ladder logic rung. Again, the ladder logic call is passing four parameters, a pointer to %R1, a pointer to %R2, a pointer to %P1, and a pointer to %P2. The C program expects six parameters, all pointers. The block will then associate each of the declared parameters to GefMain( ) with the pointers passed from the ladder logic call as follows:

```
T_INT16 *x1 refers to %R1 /* OK */  
T_INT16 *x2 refers to %R2 /* OK */  
T_INT16 *x3 refers to %P1 /* error - wrong parameter */  
T_INT16 *y1 refers to %P2 /* error - wrong parameter */  
T_INT16 *y2 refers to an unknown value on the PLC stack  
T_INT16 *y3 refers to an unknown value on the PLC stack
```

The unknown values on the PLC stack will be used for y2 and y3 and will cause the C program to write erroneously into PLC memory or cause a page fault. The exact location of the write is unpredictable.

---

**Note:** Always verify that the number of parameters expected by a block and the number the ladder logic call will pass to that block are the same. Always verify that the parameters are not NULL pointers before using.

---



## 3.6.8 Uninitialized Pointers

Use of an uninitialized C pointer variable in your C application can cause catastrophic effects on the PLC. It is essential that all pointer variables be correctly initialized prior to use by a C application.

### BAD PROGRAM - Uninitialized Pointer

```
int GefMain() {
    T_BYTE  *bad_ptr;
    T_INT16 loop;

    /* Attempt to initialize data area through */
    /* uninitialized pointer.                  */
    for (loop = 0; loop < 10; loop++) {
        *bad_ptr = 0;
    }

    return(GEF_EXECUTION_OK);
}
```

### WARNING

All pointer variables in a C application, including those used by library functions, must be initialized before they are used, or unpredictable results will occur. The use of an uninitialized pointer may result in the PACSystems logging a fatal fault in the controller fault table and going to STOP/HALT mode.

Uninitialized pointers may also result from a C block user not setting all required parameters. Check parameter pointers for NULL before using.

## 3.6.9 PLC Local Registers (%P and %L)

C Blocks have access to %P and %L PLC reference memory through several macros or functions provided in the file PACRxPlc.h in the C Toolkit. When referencing %P and %L from a block, the following two reference memories appear as two separate tables:

```
int GefMain( ) {    /* no parameters to GefMain */
                   PW(1) = RW(1);    /* Copy %R1 to %P1    */
                   LW(1) = RW(2);    /* Copy %R2 to %L1    */
                   return(GEF_EXECUTION_OK);
}
```

The PLC memory location used as %L or %P is determined by the PACSystems at runtime, based on the context from which the block was called. If the block is called from the **MAIN** ladder logic block, then all %L references inside the block will reference the %P table. The %P table and the %L table are the same when a block is called from the main block

If, however, the same block is called from a ladder logic program block other than **MAIN**, the %P and %L tables will be separate and unique in PLC memory. When the %P and %L tables are separate, all references to %L will affect only the calling block's %L table, and all references to %P will affect only the main program block's %P table.

When called from the **MAIN** ladder logic block, the following block will set %P1 equal to %R1 and then set %L1 equal to %R2:

```
GefMain( ) {      /* no parameters to GefMain */
                 PW(1) = RW(1);    /* Copy %R1 to %P1    */
                 LW(1) = RW(2);    /* Copy %R2 to %L1    */
                 return(GEF_EXECUTION_OK);
}
```

Since %L1 is actually %P1 in this case, this results in %P1 being set to the value contained in %R2. Again, this is because %P and %L, when used in a block, refer to the same memory table when called from the **MAIN** ladder logic block. Conversely, when this same block is called from any ladder sub-block, the result will be that %P1 equals %R1 and that %L1 equals %R2.

---

**Note:** Refer to Section 3.6.16, *Blocks as Timed or I/O Interrupt Blocks for an explanation of %P and %L in interrupt blocks.*

---

## %P and %L in Ladder Logic

The references %P and %L refer to two of the PLC's internal memory tables. Each of these types is word-oriented.

### Descriptions of %P and %L

Type	Description
%P	The prefix %P is used to assign program register references, which will store program data from the main program block. This data can be accessed from all program blocks. The size of the %P data block is based on the highest %P reference in all ladder logic program blocks.
%L	The prefix %L is used to assign local register references, which will store data unique to a ladder logic program block. The size of the %L data block is based upon the highest %L reference in the associated ladder logic program block.

Both %P and %L user references have a scope associated with them. Each of these references may be available throughout the logic program, or access to these references may be limited to a single ladder logic program block.

### Data Scope of %P and %L

User Reference	Range	Scope
%P	Program	Accessible from any program block.
%L	Local	Accessible from within a ladder logic block. Also accessible from any external block called by the ladder logic block.

In a program block, %P should be used for program references which will be shared with other program blocks. %L are local references which can be used to restrict the use of register data to that ladder logic program block and any C block called by that ladder logic block. These references are not available to any other parts of the program.

## 3.6.10 Block Enable Output (ENO)

In ladder logic, the function block CALL, when used with a block as the target, provides a boolean ENO output. This ENO output from the call is under the direct control of the block.

The ENO output is controlled by the return value from **GefMain()**. If **GefMain()** returns a value of GEF\_EXECUTION\_OK, the ENO output is turned ON (1). If, however, **GefMain()** returns a value of GEF\_EXECUTION\_ERROR, the CALL function block ENO output is turned OFF (0). (The C symbols GEF\_EXECUTION\_OK and GEF\_EXECUTION\_ERROR are defined in the toolkit file **PACRxPlc.h**.)

### 3.6.11 Writes to %S Memory Using SB(x)

The %S table is for the PLC to provide status on its operation. This table is intended to be written only by the CPU firmware; therefore, it is also intended to be read-only from elsewhere in the system, specifically from the application program. Attempting to use the SB(x) macro to write into %S memory will result in a compile error when compiling the application C source file. Similarly, attempting to use the pointer variable **sb\_mem** (provided in **PACRxPlc.h** and the same pointer variable used by the SB(x) macro) will result in the same compile error.

### 3.6.12 FST\_EXE and FST\_SCN Macros

In the file **PACRxPlc.h** (provided in the PACSystems C Toolkit), there are two macros, **FST\_SCN** and **FST\_EXE**, that provide blocks with direct access to %S0001 (system first scan indication) and with direct access to the block's first execution bit. The **FST\_SCN** macro references %S0001 and acts exactly like the ladder logic reference **FST\_SCN** (%S0001). If a block is not called on the first PLC sweep, the macro **FST\_SCN** should not be used for initializing data in the block. In this case, **FST\_SCN** would never be true.

The **FST\_EXE** macro operates differently than the **FST\_SCN** macro. There is no system status bit associated with the first call to blocks. A block inherits **FST\_EXE** from the block that calls it. Therefore, if **FST\_EXE** in the calling ladder logic program is true, when the block is executed, the C macro **FST\_EXE** will also be true. The value of **FST\_EXE** is determined by the calling ladder logic block, **not** by the C block. **FST\_EXE** may be TRUE (1) if the block is called multiple times from one ladder logic block or is called from multiple ladder logic blocks. If the call from the ladder logic to the block is conditional, it is possible that the block may never see **FST\_EXE** as true.

### 3.6.13 LST\_SCN Macro

The **LST\_SCN** macro provides access to the %S00002 (system last scan indication) bit. This bit is 1 when the CPU transitions to Run mode and cleared when the CPU is performing its final sweep. The CPU clears this bit (0) and then performs one more complete sweep before transitioning to Stop or Stop Faulted mode. If the number of last scans is configured to be 0, %S0002 will be cleared after the CPU is stopped and user logic will not see this bit cleared.

If a C subroutine is not called on the last scan before a PLC enters Stop mode, the **LST\_SCN** macro should not be used in that block to capture data or trigger events on the last scan. In such a case, the data or events would never be triggered because the C subroutine was not called on the last scan.

## 3.6.14 Runtime Error Handling

When a C application executes in a PACSystems CPU, if an error is generated from one of the runtime library functions or from incorrect interaction between the C application and the CPU, the error will be detected and logged in the controller fault table as an application fault on the CPU (rack 0, slot 1). Examples of such errors include, but are not limited to the following:

1. Integer divide by 0
2. Stack overflow
3. Page fault

When a runtime error is logged into the controller fault table, the fault will contain a text message describing the error.

An example of a runtime error and the resulting controller fault is illustrated in the following C application, DV0.C:

```
Example:
#include "PACRxPlc.h"

int GefMain() {
    T_INT32 x=3, y=0;

    return(x/y);
}
```

The faults logged in the CPU and displayed by Machine Edition software appear as follows:

**Fault Description:** Program runtime error

**Fault Extra Data (in ASCII format):** Div by 0

---

**Note:** On RSTi-EP EPSCPE1xx platforms, the behavior varies. Please refer Migration Notes for RSTi-EP Compatibility

---

## 3.6.15 C Application Impact on PLC Memory

As displayed on the PC, the size of a .gefElf output file is the relocate-able image... When the C application is stored to the CPU, the CPU must allocate more memory than merely the .gefElf size. The additional space allocated by the CPU includes:

1. The located executable image of the .gefElf file
2. The saved the initial values of C application global data (initialized global data)
3. Pertinent information regarding the C application (internal processing overhead)
4. A copy of the original .gefElf file.

One method of determining the PLC memory usage is to view the status dialog in the programmer and note the Program Logic usage of the folder stored without the C Block and the same folder stored with the C Block.

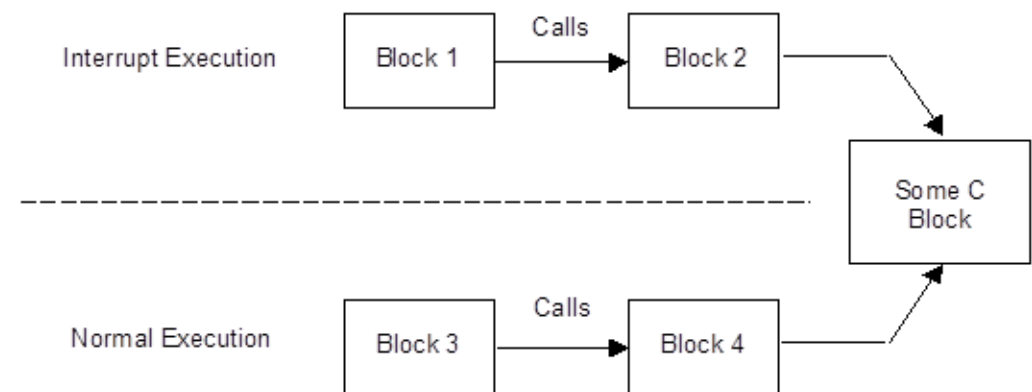
## 3.6.16 Blocks as Timed or I/O Interrupt Blocks

Blocks may be used in the PLC as the target of a timed or I/O interrupt with the following restrictions.

1. A block invoked as the result of a timed, I/O, or module interrupt may not have parameters associated with the call. The block must have 0 input parameters and 0 output parameters. A block invoked as a sub block of a timed, I/O, or module interrupt may have parameters associated with the call.
2. When a block is invoked as a timed, I/O, or module interrupt, all references to %L memory will reference the same location in the %P table. (This action is the same as when a block is called directly from the MAIN logic program.) When a block is invoked as the sub block of a timed, I/O, or module interrupt block, all references to %L memory will be references to the %L of the block from which they were called.
3. Additional interrupts are not processed while a timed, I/O, or module interrupt blocks and associated sub blocks are executing if preemptive block scheduling is disabled. The preemptive block scheduling feature is available on PACSystems firmware revision 2.0 and greater.

The following example and associated text cover the issues related to using C Blocks when the same C application is going to be called during the normal execution of the program and from a possible timed, I/O, or module interrupt.

**Figure 15: Interrupt Block Calls and C Blocks/FBKs**



In the example shown in Figure 15, two separate execution paths are depicted: normal execution and interrupt execution. Normal execution is initiated through the standard sweep mechanism of the operating system calling the `_MAIN` block. Then through a series of calls to sub-blocks, the example eventually calls “some C block”. Interrupt execution is initiated by either a timed event or by an interrupt event (interrupt input) coming into the CPU, causing the operating system to invoke a block. Note that calling a C block terminates the call chain.

The example in Figure 15 shows that both the Normal Execution path **AND** the Interrupt Execution path calling (through a chain of different blocks) the same C block. For this example, to work correctly, the C Block must be designed for re-entrant operation.

A C Block developer should use the following guidelines to ensure the success of a situation such as the one illustrated in Figure 15.

1. All variables used by the C Block should be stack-based (automatic) variables.
2. If there is any portion of the Block that operates on PLC global memories (%R, %P, ...etc.), the Block must contain additional code to handle some sort of hand-shaking between normal executions and interrupt executions to prevent data incoherency. The hand-shaking could be accomplished by declaring a global flag (variable) in the Block (or using an application-reserved location in PLC global memory) that the Block sets just prior to writing to the PLC global memories and then clears when the update is complete. Execution of the block (regardless of normal or interrupt) should read the global flag before changing the PLC global memory. If the flag is set, the C Block should not change the PLC global memory.
3. Use re-entrant versions of functions.

### 3.6.17 Restricting Compilation to a Specific Target

In most cases, you will want to use the PACRXPlc.h header file and the corresponding command line

**compileCPACRx**

to compile a C Block for any PACSystems RX PLC. If you want to compile your application for a specific target (such as the RX7i or RX3i), you can use the command line

**compileCPACRx7i**

or

**compileCPACRx3i**

respectively while still using the PACRXPlc.h header file.

However, if you always want to restrict compilation for a specific target on a particular C Block, you should use the PACRX7iPlc.h for the RX7i target or PACRX3iPlc.h for the RX3i target. By using these header files, the C Block will successfully compile only for the specified target. For example, if you use the "PACRX7iPlc.h" header file in your C Block source file, you must use the

**compileCPACRX7i**

command line to successfully compile the C Block. In this case if you attempt to use the

**compileCPACRX3i**

command line, the compilation will fail.

If you want to compile your application for RSTi-EP target (such as EPSCPE100 or EPSCPE115), you should use the

**compileCPACRSTi**

command line, while using the PACRSTiPlc.h header file.

---

**Note:**

- *As of Release 3.5, C Block functionality between Rx3i and Rx7i targets is essentially the same so that compilation for specific targets is currently not needed. However, the PACSystems C Toolkit is structured to support variation between targets in case it is needed in the future.*
  - *compileCPACRX7i and compileCPACRX3i are compatible with 32-bit C Blocks only; compileCPACRx is the only command compatible with 64-bit C Blocks.*
-



# Chapter 4: Debugging and Testing C Applications

## 4.1 Testing C Applications in the PC Environment

It is highly recommended that all C applications be tested prior to execution on the PACSystems CPU. This is best accomplished by testing the application on the PC using the PC debugging environment provided by the C Toolkit. This environment provides various batch files that use the Cygwin compiler, linker and debugger to produce an \*.exe file that can be directly executed on the PC. The first step is to develop the C Block source code using the editor of your choice. The C Toolkit provides a set of stub functions for each of the C Toolkit PLC functions that are compiled and linked to your C Block during the PC compilation process. These stub functions are located in the "Targets\CommonFiles\TargetStubLibCommon" subdirectory. You can modify these stub functions to simulate dynamic behavior.

When debugging on the PC, the C Run-Time library functions of the Cygwin environment are used. However, some non-standard C library functions, such as reentrant forms of functions such as `div_r()` for `div()`, must be used on the PLC. These functions are provided by the C Toolkit and compiled and linked with your C Block during the PC compilation process. After compiling and linking the C block, you can then run the application using the Cygwin environment to simulate and debug the application.

The following steps describe how to debug an application on the PC:

1. Create C Test driver code that initializes memory pointers and calls the C Block to be tested. An example is given below:

```
/* C PC Driver code - ctkPcDriver.c */  
  
#include "PACRXPlc.h" /* For any PACSystems PLC */  
  
/* For RX3i use PACRX3iPLC.h For Rx7i use PACRx7iPLC.h */  
  
#include "ctkInitCBlock.h"  
  
  
/* declare GefMain as external function in another file */  
extern int GefMain(T_WORD *pR8, T_BYTE *pI1000, T_BYTE *pM500);  
  
  
int main(int argc, char *argv[])  
{  
    initCBlock(); /* creates ref mem and initializes pointers to that
```

```

        memory*/
    GefMain(&RW(8), &Ib(1000), &Mb(500)); /* calling main passing
        pointers to %R8, %I1000 and
        %M500 */

    return 0;
}

```

To avoid having to remove or bypass this code when compiling for the PLC, it is recommended that this code reside in another C source file and then compiled with the C Block under test.

2. Create your C Block application. An example is shown below:

```

/* myCBlock.c */
#include <stdio.h>
#include <PACRXPlc.h>
T_INT32 status;
T_INT32 failCount = 0;

int GefMain(T_WORD *pR8, T_BYTE *pI1000, T_BYTE *pM500)
{
    if (*pR8 != 0)
    {
        RW(10) = * pR8; /*write %R8 to %R10 as word */
        RD(12) = failCount;
        return GEF_EXECUTION_OK;
    }
    else
    {
        *pM500 = *pI1000; /* Copy %I1000 (one byte) to %M500) */
        status = GEF_EXECUTION_ERROR;
        failCount++;
        return status;
    }
}

```

3. Optionally add code to the PLC C stub functions to simulate the desired PLC behavior.

---

**Note:** *PLC C stub function files are located in the following directory:*

*<yourInstallDir>\PACSystemsCToolkit\Targets\CommonFiles\TargetStubLibCommon*

---

4. Create sourcesDebug file that specifies which files to compile together: An example is given below:

---

**Note:** *Comments can be included by putting a "#" in the first column.*

---

**# sourcesDebug file**

**CFILENAMES = myCBlock.c ctkPcDriver.c**

5. Start the C Toolkit. (Double click the desktop icon (PACSystems(TM) C Toolkit) or use the Start menu to execute the file ctkPACS.bat located at the Toolkit's root directory.)
6. Within the DOS window created in step 4, compile the C Block for the PC using the following command in the same directory containing the source files and the sourcesDebug file:

**CompileCDebugPACRX**

**(For Rx7i targets, use CompileCDebugPACRX7i; for RX3i targets, use CompileCDebugPACRX3i, for RSTi-EP targets, use CompileCDebugPACRSTi)**

7. Run the Cygwin debugger using the following command:

**debugPACRX pc\myCBlock.exe**

**(For Rx7i targets, use DebugPACRX7i; for RX3i targets, use DebugPACRX3i, for RSTi-EP targets, use DebugPACRSTi)**

This will bring up a Windows based debugger that allows setting break points, single step, viewing and changing memory, etc. Use the help facility within this application for information on how to use the debugger.

### CAUTION

The Toolkit places copies of the PLCC stub file source code into the "\pc" directory to allow source line debugging. You should not modify these files because they will be replaced with the master stub files located in "Targets\CommonFiles\TargetStubLibCommon" each time your source files are recompiled.

- 
8. The C Block can also be run at the DOS prompt with the following command:

**runPACRX pc\myCBlock.exe**

**(For Rx7i targets, use runPACRX7i pc\myCBlock.exe; for RX3i targets, use runPACRX3i pc\myCBlock.exe; for RSTi-EP targets, use runPACRSTi pc\myCBlock.exe)**

Debugging in this case requires `PLCC_MessageWrite()` statements within the application to indicate program flow and state.

## 4.2 Debugging C Applications in the PLC

There are two primary ways to debug the C application operating in the PLC: message mode writes to serial port and reference table monitoring.

### 4.2.1 Message Mode Debugging

The use of **PLCC\_MessageWrite** to debug a C application running in a PACSystems is very similar to using **PLCC\_MessageWrite** to debug the same C application on the PC. The **PLCC\_MessageWrite** statements should be placed in the source code to provide a road map of the execution path and to display the value of any key data items.

---

**Note:** For `PLCC_MessageWrite` to work, the CPU's serial port must be configured for Message mode. If the CPU's serial port is not configured for Message mode and `PLCC_MessageWrite` is called, no characters are placed into the print queue and the return value from `PLCC_MessageWrite` is -1.

---

### 4.2.2 Reference Table Monitoring

As with **PLCC\_MessageWrite** debugging, the execution path and key data items may be determined by modifying a C application to place this information into unused areas of the global PLC reference tables (%R, %W, %M, %T, %P, etc.) and then viewing the saved execution road map and key data items through the programmer's online reference display(s).

# Chapter 5: Conversion Notes and Series 90 Compatibility

For the most part, C Block programming with the PACSystems is very similar to the Series 90-70 and Series 90-30 PLCs. This chapter describes differences that must be considered when converting Series 90-70 or Series 90-30 applications to PACSystems. C blocks in existing Series 90 program folders must be recompiled using the PACSystems C Toolkit.

## 5.1 Series 90 Compatibility Header Files (PLCC9070.h and PLCC9030.h)

To minimize conversion issues when converting Series 90 applications, use the appropriate include file in your C Block application:

Series 90-70	Use PLCC9070.h instead of PACRXPLC.h or PACRX7iPlc.h.
Series 90-30	Use PLCC9030.h instead of PACRXPLC.h or PACRX3iPlc.h. If a C block is used as the _MAIN block in a 90-30 folder, you must compile the C source into a program block and create a one-rung main program in LD that calls this block.

If a C block is used as the \_MAIN block in a 90-30 folder, you must compile the C source into a program block and create a one-rung main program in LD that calls this block.

### PLCC9070.h

This file equates some of the 90-70 C Toolkit names to the equivalent PACSystems C Toolkit names. For example, in the 90-70 C Toolkit many run-time functions have a “far” version such as \_fstrcat. Since the PACSystems does not require the **far** version of functions, the PLCC9070.h file equates them to the non-far function such as strcat for \_fstrcat. Similarly, the 90-70 C Toolkit used OK and ERROR as defines for the return value that controls the state of ENO. These are equated to GEF\_EXECUTION\_OK and GEF\_EXECUTION\_ERROR respectively. Also, this file equates some of the common basic types such as byte and word to the equivalent PACSystems types, T\_BYTE and T\_WORD.

### PLCC9030.h

This file equates some of the 90-30 C Toolkit names to the equivalent PACSystems C Toolkit names. Similar to the 90-70 version, the PLCC9030.h file equates far versions of functions to non-far versions. This file also equates common basic types such as byte and word to the equivalent PACSystems types, T\_BYTE and T\_WORD.

**For new applications, one of the following target header files should be used:**

PACRXPLC.h	Compiles C Blocks that work with all PACSystems CPU targets.
PACRX3iPLC.h	Compiles C Blocks that work with PACSystems RX3i target.
PACRX7iPLC.h	Compiles C Blocks that work with PACSystems RX7i target.
PACRSTiPlc.h	Compiles C Blocks that work with PACSystems RSTi-EP target.

## 5.2 Writing Directly to Discrete Memory

If the application uses the Series 90 style macros that write directly to discrete reference memory (%I, %Q etc.), overrides will not be respected, and the corresponding transition bit will not be set because this functionality is not implemented in hardware on the PACSystems product. Since there is not a one-to-one correspondence in the functionality of the Series 90, and PACSystems discrete macros, the PACSystems discrete macro definitions are similar to the Series 90 macros, but slightly different to flag potential overrides and transition issues. For example, the macro that accessed a byte of %I memory in Series 90 PLCs was called IB(). In the PACSystems C Toolkit, it is called Ib(). If you want to overrides to be respected and to set the corresponding transition bit, you must use a set of new read/write PLC functions. Here are some compatibility/conversion examples:

- a. **Direct assignment to discrete reference.** Here is an example of Series 90 C code:

```
IB(1) = 0x33;
```

Here is how the code must be written for the PACSystems to have the same functionality as the Series 90:

```
WritePlcByte(I_MEM, 1, 0x33, FALSE);
```

The first parameter of the WritePlcByte function determines which reference table to access. The second parameter of determines the reference address to access. The third parameter determines the value to write to the reference table. The fourth parameter determines if the byte is written to the most or least significant byte if using a word reference. Since the write occurs to a discrete reference the parameter is unused. If the “RefTable” or “address” are out of range, no reference memory values are changed, and the function returns GEF\_ERROR. If the “RefTable” and “address” are within range, the function returns GEF\_OK. The prototype for this function is shown below:

```
T_INT32 WritePlcByte(T_WORD RefTable, T_DWORD address,  
                    T_BYTE writeValue, T_BOOLEAN msbByte);
```

- b. **Reading a discrete reference.** Here is an example of Series 90 C code:

```
MyVar1 = IB(1);
```

Because this is a read operation that does not need to take into account override and transition bits, you have the choice of using a macro or a function call to get the same functionality as the Series 90 PLC.

**Macro:**

```
MyVar1 = Ib(1);
```

**Function Call:**

```
MyVar1 = ReadPlcByte(I_MEM, 1, FALSE);
```

- c. **Using test bit, bit set or bit clear functions.** In this case, there is no coding change from Series 90 to the PACSystems because a function is implemented using the same syntax as the macro. The function carries out the proper behavior with

respect to overrides and transition bits. From a reuse standpoint, the macro call looks exactly the same as the function call so there is no coding change required. For example, the following 90-70 C code does not need to change:

```
if (BIT_TST_I(1))
{
    BIT_SET_I(2);
}
else
{
    BIT_CLR_I(2);
}
```

- d. **Other Macros.** Most other macros can be used just as they were used in the Series 90 PLC and require no conversion. A complete list of all macros and their compatibility with the 90 70 and 90-30 macros is located in Appendix A:.

## 5.3 PLC Target Library Function Compatibility Issues

Most 90-70 and 90-30 Target Library functions are supported but there are some compatibility issues. A complete list of all PACSystems Target Library functions and compatibility issues are described in Appendix A.

## 5.4 Compatibility Issues with Retentive Global Variables

In the Series 90, C Block's retentive global variables are uninitialized and denoted with the "static" attribute. All other global variables are non-retentive. Although not documented in the Series 90, uninitialized non-retentive global variables were set to 0 on a run to stop transition. For the PACSystems C Blocks, both static and non-static uninitialized global variables are retentive and are truly uninitialized (not set to 0). Users who relied on uninitialized non-static variables being set to 0 on a stop to run transition will need to add initialization code. PACSystems C Blocks with initialized variables are non-retentive which is the same behavior as the Series 90.

## 5.5 “int” Type Issues

The “int” basic type in the 90-70 and 90-30 represents a 16-bit signed number. However, PACSystems is a 32-bit system so the “int” basic type is a 32-bit signed number. You will need to evaluate your programs to determine if this conversion causes any issues. Here are some examples:

```
int myVar;  
  
myVar = RI(1); /* sets myVar to equal %R1 as a 16-bit signed value */
```

When this is compiled and executed on a PACSystems PLC, the least significant 16 bits will be set equal to %R1. The most significant 16 bits will be set to 0 unless the number is negative in which case the most significant 16 bits will be set to 0xffff (sign extended). This case should not typically cause any problems because the cast is from a smaller to a higher number of bits.

However, the reverse case may cause problems in some cases.

```
int myVar;  
  
RI(1) = myVar; /* sets %R1 with a 32-bit signed value */
```

In this case, the least significant 16 bits of myVar will be written to %R1. Thus, if the value of myVar is outside the range of a signed 16-bit number (+32767 to -32768), then the value in %R1 will be a signed 16-bit truncated version of myVar. For example, if myVar is 32768 (0x00008000), the value in %R1 will be -32768 (0x8000).

## 5.6 “enum” Type Issues

The “enum” basic type in the 90-70 and 90-30 represents a 16-bit signed number. However, PACSystems is a 32-bit system so the “enum” basic type is a 32-bit signed number. You will need to evaluate your programs to determine if this conversion causes any issues.

## 5.7 Non-Standard C Library Functions

Non-Standard C Library functions are not supported in the PACSystems C Toolkit. See appendix B for functions that are not supported.

## 5.8 Entry Point

In the Series 90 C Toolkit, the entry point into the user application was main(). For the PACSystems C Toolkit, the entry point is GefMain().

## 5.9 C Standalone Programs

C standalone programs are not supported. However, C program applications that do not rely on the C program scheduling features can be compiled and executed as C blocks.



## 5.10 Use of Input Parameters as Pointers to Discrete Memory Tables

In the PACSystems C Toolkit if the user application is passed a pointer to one of the discrete memory tables (%I, %Q etc.), for example as one of the input parameters to GefMain(), and the pointer is used to write to discrete reference table memory, overrides and transitions are not taken into account for the write operation. When a discrete memory write operation occurs via a pointer in the Series 90 PLCs, overrides and transitions are taken into account.

For the PACSystems C Toolkit, you should use the following function when writing directly to discrete memory via a pointer if you want overrides and transition bits to be respected:

```
T_INT32 PlcMemCopy(void *pDestination, void *pSource, T_DWORD size);
```

## Chapter 6: Installed Sample Blocks

In the C Toolkit directory structure, there are two subdirectories under the Projects directory that contain examples of blocks, SampleProj1 and SampleProj2.

### 6.1 SampleProj1

The SampleProj1 directory contains three sample C files that generate a C Block from a single C source file. Each file is discussed below:

- `ctkCBlockTest.c` is intended for compilation for the RX7i, RX3i, or PACRX and makes a call to every function and macro supported by the C Toolkit. This block is an example of an application without input/output parameters.

Because `ctkCBlockTest` exercises all available toolkit routines and macros, it will not execute on a PACSystems CPU with the default configuration. See the setup note at the top of the C file for more information.

- `ctkCBlockTestParams_7_7.c` provides a simple example using seven input and seven output parameters. The application equates the output to the inputs, simulating a simple move type of operation. In addition, it provides an example of controlling ENO by returning `GEF_EXECUTION_ERROR` (ENO off) if input 1 (`pCoolantTemp` is greater than 1000) or `GEF_EXECUTION_OK` (ENO on) otherwise.

To execute this sample block on a PACSystems CPU, the C block must be setup as a parameterized block with 7 WORD inputs and 7 WORD outputs.

- `ctkCBlockTestPc.c` is a version similar to `ctkCBlockTest.C` with additional driver code at the end of the file so that it can be compiled and run on the PC.

## 6.2 SampleProj2

The SampleProj2 directory contains an example for compiling multiple C sources into a single C Block. The files to be compiled and linked together for the PLC execution are specified in the “sources” file. Similarly, the files to be compiled and linked together for PC debugging are specified in the sourcesDebug file. This directory also has examples of precompiled object files.

1. ctkCBlockTest4.plc0, ctkCBlockTest5.plc0, and ctkCBlockTest.plc0 for PLC linking.
2. ctkCBlockTest4.pc0, ctkCBlockTest5.pc0, ctkCBlockTest.pc0 for PC linking.

These files were produced by compiling their corresponding C source file with the following command for the PLC object files:

**compileCPACRX7i ctkCBlockTest4 DisableGefLibLink**

And the following command for the PC object files:

**compileCDebugPACRX7i ctkCBlockTest4 DisableGefLibLink**

The “sources” and “sourcesDebug” files respectively specify the use of these object files for compilation as opposed to the source file.

All files are compiled and linked together with one of the following commands for the PLC:

**compileCPACRX7i**

**compileCPACRX3i**

**compileCPACRX**

**compileCPACRSTi**

And one of the following commands for the PC:

**compileCDebugPACRX7i**

**compileCDebugPACRX3i**

**compileCDebugPACRX**

**compileCDebugPACRSTi**

These files also illustrate the use of the serial port message mode read/write functions.

This sample block will not execute on a PACSystems CPU with the default configuration. See the setup note at the top of ctkCBlockTest1.c for more information.

# Chapter 7: Migration Notes for RSTi-EP Compatibility

For the most part, C Block programming with the RSTi-EP is very similar to the RX3i and RX7i PLCs. This chapter describes differences that must be considered when converting RX3i and RX7i applications to RSTi-EP platforms. C blocks must be recompiled using the PACSystems\_RSTi-EP\_EPSCPE1xx\_C\_Toolkit\_Rel8.10.

## 7.1 C Block Executable memory considerations

A PACSystems C block is, by default, limited to 256Kbytes in size. The EPSCPE100/115 platforms support 1 MB and 1.5 Mb user memory respectively. The overall program logic including C Blocks should not exceed these memory boundaries. Otherwise the PAC Machine Edition will abort the downloading of hardware configuration to the target.

## 7.2 Notes on unaligned memory access in C Application

Whenever the C Block application accesses/operates on unaligned user memory, the CPU logs a fatal fault (Program Runtime Error with fault extra data “Data Abort” along with the C Block name) in the controller fault table and goes to STOP/HALT mode.

**Figure 16: Unaligned memory access fatal fault on RSTi-EPSCPE1xx platform**

0.0	Program runtime error			01-01-2010 05:10:37
	Error Code	Group	Action	Task Num
	17	22	3:Fatal	0
	Fault Extra Data: Data abort.....MACROS.P			

Please refer to sections 7.3, 7.4 & 7.5 for further details which explains about the memory unalignment in the C application.

### Recovery from HALT mode

To recover CPE100/115 from STOP-HALT mode, press and hold the membrane switch and power down the CPE100/115.

**Note:** During this operation RAM contents gets erased.

For further information on halt mode recovery please refer to the following documents.

GFK-3039\_EPSCPE115\_Quick Start Guide

GFK-3012\_EPSCPE100\_Quick Start Guide

## 7.3 Limitations on Double/float discrete data type macros

### ⚠ CAUTION

On RSTi-EP EPSCPE1xx platforms, the double precision floating point macros such as RDbI, AiDbI, AQDbI, WDbI, PDbI, LDbI and single precision floating point macros such as RF, AIF, AQF, WF, PF, LF should only be used for reads but NOT for writes. A safer alternative is to use read/write PLC functions that perform address boundary checking. These functions are ReadPlcDouble, WritePlcDouble.

Below given are some examples:

- a. **Direct assignment to discrete reference.** Below is an example of C code:

```
RDbI(1) = 0x01;
```

Which results in compilation error as:

Figure 17:

```
memoryLR.c: In function 'GefMain':
memoryLR.c:6:2: warning: implicit declaration of function 'RDbI' [-Winimplicit-function-declaration]
  RDbI(1) = 0x01;
  ^
memoryLR.c:6:18: error: lvalue required as left operand of assignment
  RDbI(1) = 0x01;
  ^
GNUmake: *** [memoryLR.gef0] Error 8x1
Warning: If link errors are reported above, the following problems may exist
and may prevent storing the C Block to the PLC:
1) The Toolkit has detected C Block functions which potentially are not
supported by the PLC. Check user documentation for functions supported by
the intended PLC.
2) The Toolkit has detected functions which might be declared in another
user C source file. In this case, perform a multiple file compilation by
including all associated C source files in the "sources" file. See
user documentation for details.
3) The Toolkit has detected the lack of a GefMain function in the user
application.
If compile errors are reported above, they will be repeated below and must
be fixed before storing the C Block to the PLC.
Link and compile errors are recorded in a file called "build.log".
```

Given is how the code must be written for the RSTi-EP platform to have the same functionality:

```
WritePlcDouble(R_MEM, 1, 0x33);
```

- b. **Reading** a discrete reference. Given is an example of C code:

```
double MyVar1 = RDbI(1);
```

Because this is a read operation that does not need to consider override and transition bits, you have the choice of using a macro or a function call to get the same functionality as the RX3i and RX7i PLC.

**Macro:**

```
double MyVar1 = RDbI(1);
```

**Function Call:**

```
double MyVar1 = ReadPlcDouble(R_MEM, 1);
```

## 7.4 Limitations on typecasting of double data type

Wherever T\_REAL64 data type variables, type casting has been used in the code that copies data from unaligned memory such as Byte/Half word boundary to the above data type, results in a CPU STOP/HALT condition.

### **⚠ WARNING**

All double data type pointer type conversions in a C application, must be done using memcpy() standard library function. Otherwise use of these assignments may put the controller in STOP/HALT condition with a fatal fault (Program Runtime Error with fault extra data “Data Abort” along with the C Block name) logged into the controller fault table.

#### Example:

Existing C Block	Mandatory change
<pre>int GefMain () {     .....     T_BYTE val = 12;     T_REAL64 destinationLoc     T_BYTE* pParamValue = &amp;val;     destinationLoc = *( T_REAL64*)     pParamValue;     ..... }</pre>	<pre>int GefMain () {     .....     T_BYTE val = 12;     T_REAL64 destinationLoc     T_BYTE* pParamValue = &amp;val;     memcpy(&amp;destinationLoc, pParamValue,     sizeof(T_REAL64));     ..... }</pre>

## 7.5 Notes on double data type pointer assignment limitations

### WARNING

All double pointer assignment operations in a C application, must be done using memcpy( ) standard library function. Otherwise use of these assignments may put the controller in STOP/HALT condition with a fatal fault (Program Runtime Error with fault extra data “Data Abort” along with the C Block name) logged into the controller fault table.

#### Examples:

Existing C Block	Mandatory change
<pre>int GefMain (T_REAL64 *LREAL_In, T_REAL64 *LREAL_Out) {     .....     *LREAL_Out=*LREAL_In;     ..... }</pre>	<pre>int GefMain () {     .....     memcpy(LREAL_Out, LREAL_In, sizeof(T_REAL64));     ..... }</pre>
<pre>int GefMain (T_REAL64 *LREAL_Array_In, T_REAL64 *LREAL_Array_Out;) {     .....     *LREAL_Array_Out=*LREAL_Array_In;     ..... }</pre>	<pre>int GefMain () {     .....     memcpy(LREAL_Array_Out, LREAL_Array_In, sizeof(T_REAL64));     ..... }</pre>

## 7.6 Notes on usage of “ / “ and ” %” operators in C Application

### ⚠CAUTION

On RSTi-EP EPSCPE1xx platforms, the division ( / ) and modulus (%) operators are not supported. An alternative is to use the functions `div_r()` and `fmod()` for division and modulus operations respectively.

Whenever the C Block application uses division ( / ) or modulus (%) operators, it will result in compilation error as shown in the below screenshot:

Figure 18:

```
test.gef0: In function 'GefMain':
test.c:(.text+0xb0): undefined reference to '__aeabi_idiv'
GNUmake: *** [test.sm] Error 0x1
```

Example:

Existing C Block	Mandatory change
<pre>int GefMain () { float quotient; ..... quotient = dividend / divisor; ..... }</pre>	<pre>int GefMain () { div_t div; float quotient; ..... div_r(dividend, divisor,&amp;div); quotient = div.quot; ..... }</pre>

Divide by Zero:

When a C Block performs the divide by zero operation, results the maximum positive value as quotient (i.e. 2,147,483,647 (0x7FFF FFFF)).



# Appendix A: Target Library Functions

As a general note, the following functions will set `errno` in the current context, if the function does not return status in some form. `errno` contains an error code from the last Target Library or C Run Time Library function which encountered an error. You can access `errno` via the function `PLCC_GetErrno()`.

## A-1 Target Library Reference Memory Functions and Macros

Implemented in `ctkRefMem.h`

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
<code>BIT_TST_I(address);</code>	Macro compatible with 90-70 and 90-30.
<code>T_INT32 BIT_SET_I(T_DWORD address);</code> <code>T_INT32 BIT_CLR_I(T_DWORD address);</code>	Same functionality as 90-70 and 90-30 but implemented as a function rather than a macro to respect overrides and to change corresponding transition bits. This function also returns an <code>GEF_OK</code> status if the address is within range and an <code>GEF_ERROR</code> status if the address is not within range. In the <code>GEF_ERROR</code> case, the bit is not changed. Errnos: <b>TLIB_ERRNO_OFFSET_RANGE_ER</b> (Address is outside of valid range).
<code>BIT_TST_Q(address);</code>	Macro compatible with 90-70 and 90-30.
<code>T_INT32 BIT_SET_Q(T_DWORD address)</code> <code>T_INT32 BIT_CLR_Q(T_DWORD address)</code>	Function compatible with 90-70 and 90-30. Same return status as <code>BIT_SET_I()</code> .
<code>BIT_TST_M(address)</code>	Macro compatible with 90-70 and 90-30.
<code>T_INT32 BIT_SET_M(T_DWORD address)</code> <code>T_INT32 BIT_CLR_M(T_DWORD address)</code>	Function compatible with 90-70 and 90-30. Same return status as <code>BIT_SET_I()</code> .
<code>BIT_TST_T(address)</code>	Macro compatible with 90-70 and 90-30.
<code>T_INT32 BIT_SET_T(T_DWORD address)</code> <code>T_INT32 BIT_CLR_T(T_DWORD address)</code>	Function compatible with 90-70 and 90-30. Same return status as <code>BIT_SET_I()</code> .
<code>BIT_TST_G(address)</code>	Macro compatible with 90-70 and 90-30.
<code>T_INT32 BIT_SET_G(T_DWORD address)</code> <code>T_INT32 BIT_CLR_G(T_DWORD address)</code>	Function compatible with 90-70 and 90-30. Same return status as <code>BIT_SET_I()</code> .
	<code>BIT_TST_GA(x)</code> not supported (90-70 only)
	<code>BIT_SET_GA(x)</code> not supported (90-70 only)

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
	BIT_CLR_GA(x) not supported (90-70 only)
	BIT_TST_GB(x) not supported (90-70 only)
	BIT_SET_GB(x) not supported (90-70 only)
	BIT_CLR_GB(x) not supported (90-70 only)
	BIT_TST_GC(x) not supported (90-70 only)
	BIT_SET_GC(x) not supported (90-70 only)
	BIT_CLR_GC(x) not supported (90-70 only)
	BIT_TST_GD(x) not supported (90-70 only)
	BIT_SET_GD(x) not supported (90-70 only)
	BIT_CLR_GD(x) not supported (90-70 only)
	BIT_TST_GE(x) not supported (90-70 only)
	BIT_SET_GE(x) not supported (90-70 only)
	BIT_CLR_GE(x) not supported (90-70 only)
BIT_TST_SA(address)	Macro compatible with 90-70 and 90-30.
T_INT32 BIT_SET_SA(T_DWORD address)	Function compatible with 90-70 and 90-30. Same return status as BIT_SET_I().
T_INT32 BIT_CLR_SA(T_DWORD address)	
BIT_TST_SB(address)	Macro compatible with 90-70 and 90-30.
T_INT32 BIT_SET_SB(T_DWORD address)	Function compatible with 90-70 and 90-30. Same return status as BIT_SET_I().
T_INT32 BIT_CLR_SB(T_DWORD address)	
BIT_TST_SC(address)	Macro compatible with 90-70 and 90-30.
T_INT32 BIT_SET_SC(T_DWORD address)	Function compatible with 90-70 and 90-30. Same return status as BIT_SET_I().
T_INT32 BIT_CLR_SC(T_DWORD address)	
BIT_TST_R(address, bitPosition)	Macros compatible with 90-70 and 90-30.
BIT_SET_R(address, bitPosition)	
BIT_CLR_R(address, bitPosition)	
BIT_TST_AI(address, bitPosition)	
BIT_SET_AI(address, bitPosition)	
BIT_CLR_AI(address, bitPosition)	
BIT_TST_AQ(address, bitPosition)	
BIT_SET_AQ(address, bitPosition)	
BIT_CLR_AQ(address, bitPosition)	
BIT_TST_P(address, bitPosition)	Macros compatible with 90-70.

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
BIT_SET_P(address, bitPosition)	
BIT_CLR_P(address, bitPosition)	
BIT_TST_L(address, bitPosition)	
BIT_SET_L(address, bitPosition)	
BIT_CLR_L(address, bitPosition)	
BIT_TST_S(address)	Macros compatible with 90-70 and 90-30.
BIT_TST_W(address, bitPosition)	New Macros to access %W memory. Not compatible with Series 90.
BIT_SET_W(address, bitPosition)	
BIT_CLR_W(address, bitPosition)	
T_INT32 setBit (T_WORD RefTable, T_DWORD offset, T_WORD bitNumber)	<p>New function to generically set a bit reference memory. The bitNumber is only used for word type memory. The function returns GEF_OK if the bit is set and GEF_ERROR if the bit cannot be set.</p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b> (offset is outside of valid range).</p> <p><b>TLIB_ERRNO_READ_ONLY_ER</b></p> <p><b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
T_INT32 clearBit (T_WORD RefTable, T_DWORD offset, T_WORD bitNumber)	<p>New function to generically clear a bit in reference memory. The bitNumber is only used for word type memory. The function returns GEF_OK if the bit is set and GEF_ERROR if the bit cannot be set.</p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b> (offset is outside of valid range).</p> <p><b>TLIB_ERRNO_READ_ONLY_ER</b></p> <p><b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
Ib(address)	Implemented as macro compatible with 90-70 and 90-30 syntax, with the exception that the name has been changed from IB to Ib. However, this macro does not respect overrides and does not set corresponding transition bits, so the functionality is different than the 90-70 and 90-30. You should use the WritePlcByte() function to get the same functionality as the 90-70 and 90-30. (See next item.)

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
T_INT32 WritePlcByte (T_WORD RefTable, T_DWORD offset, T_BYTE writeValue, T_BOOLEAN msbByte)	<p>This function writes to reference memory taking into account overrides and transition bits. The reference memory in the specified Reference Table (RefTable) and at the specified “offset” is written using the value of “writeValue”. If the “offset” is out of range, no reference memory values are changed, and the function returns GEF_ERROR. If the offset is within range, the function returns GEF_OK. msbByte determines whether the MSB or LSB of a word type reference is written.</p> <p><b>Note:</b> <i>this function will only affect the transition bits that actually change.</i></p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b>  <b>TLIB_ERRNO_READ_ONLY_ER</b>  <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
T_BYTE ReadPlcByte (T_WORD RefTable, T_DWORD offset, T_BOOLEAN msbByte)	<p>The reference memory in the specified Reference Table (RefTable) and at the specified “offset” is read and returned by the function. Errno is set if there is an error reading the value. msbByte determines whether the MSB or LSB of a word type reference is read.</p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b>  <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
Qb(address)	Similar issues as Ib()
Mb(address)	Similar issues as Ib()
Tb(address)	Similar issues as Ib()
Gb(address)	Similar issues as Ib()
	GAB(x) not supported.
	GBB(x) not supported
	GCB(x) not supported
	GDB(x) not supported
	GEB(x) not supported
Sb(address)	Similar issues as Ib(); This is read-only, and the compiler will issue an error if you attempt to write to this memory using this macro.
SAb(address)	Similar issues as Ib()
SBb(address)	Similar issues as Ib()

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
SCb(address)	Similar issues as lb()
RB(address, highByte)	Macros compatible with 90-70 and 90-30.
AIB(address, highByte)	
AQB(address, highByte)	
PB(address, highByte)	Macros compatible with 90-70.
LB(address, highByte)	
WB(address, highByte)	New Macro to support %W memory. Not supported by Series 90.
lw(address)	Implemented as macro compatible with 90-70 and 90-30 syntax with the exception that the name has been changed from IW to lw. However, this macro does not respect overrides and does not set corresponding transition bits, so the functionality is different than the 90-70 and 90-30. You should use the WritePlcWord() function to get the same functionality as the 90-70 and 90-30.(see next item).
T_INT32 WritePlcWord (T_WORD RefTable, T_DWORD offset, T_WORD writeValue)	<p>This function writes to reference memory taking into account overrides and transition bits. A word (16 unsigned bits) of reference memory in the specified Reference Table (RefTable) and at the specified “offset” is written with the “writeValue”. If the “offset” is out of range, no reference memory values are changed, and the function returns GEF_ERROR. If the “offset” is within range, the function returns GEF_OK.</p> <p><b>Note:</b> <i>this function will only affect the transition bits that actually change.</i></p> <p>Errnos:</p> <p>TLIB_ERRNO_OFFSET_RANGE_ER TLIB_ERRNO_READ_ONLY_ER TLIB_ERRNO_INVALID_REF_TABLE_ER</p>
T_WORD ReadPlcWord (T_WORD RefTable, T_DWORD offset)	<p>A word (16 unsigned bits) of reference memory in the specified Reference Table (RefTable) and at the specified “offset” is read and returned by the function. Errno is set if there is an error reading the value.</p> <p>Errnos:</p> <p>TLIB_ERRNO_OFFSET_RANGE_ER TLIB_ERRNO_INVALID_REF_TABLE_ER</p>
Qw(address)	Similar issues as lb()

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
Mw(address)	
Tw(address)	
Gw(address)	
	GAW(x) not supported.
	GBW(x) not supported.
	GCW(x) not supported.
	GDW(x) not supported.
	GEW(x) not supported.
Sw(address)	Similar issues as lb(). This is read-only and the compiler will issue an error if you attempt to write to this memory using this macro.
SAw(address)	Similar issues as lb()
SBW(address)	Similar issues as lb()
SCw(address)	Similar issues as lb()
RW(address)	Macro Compatible with 90-70 and 90-30.
AIW(address)	
AQW(address)	
PW(address)	Macro Compatible with 90-70.
LW(address)	
WW(address)	New Macro to support %W memory. Not supported by Series 90.
li(address)	Implemented as macro compatible with 90-70 and 90-30 syntax with the exception that the name has been changed from ll to li. However, this macro does not respect overrides and does not set corresponding transition bits, so the functionality is different than the 90-70 and 90-30. You should use the WritePlcInt() function to get the same functionality as the 90-70 and 90-30. (see next item).

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
T_INT32 WritePlcInt (T_WORD RefTable, T_DWORD offset, T_INT16 writeValue)	<p>This function writes to reference memory taking into account overrides and transition bits. Reference memory in the specified Reference Table (RefTable) and at the specified “offset” is written with the “writeValue” as a 16-bit signed integer. If the “offset” is out of range, no reference memory values are changed, and the function returns GEF_ERROR. If the offset is within range, the function returns GEF_OK.</p> <p><b>Note:</b> <i>this function will only affect the transition bits that actually change.</i></p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b> <b>TLIB_ERRNO_READ_ONLY_ER</b> <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
T_INT16 ReadPlcInt (T_WORD RefTable, T_DWORD offset)	<p>Reference memory in the specified Reference Table (RefTable) and at the specified “offset” is read as a 16-bit signed integer and returned by the function. Errno is set if there is an error reading the value.</p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b> <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
Qi(address)	Similar issues as Ib().
Mi(address)	
Ti(address)	
Gi(address)	
	GAI(x) not supported
	GBI(x) not supported
	GCI(x) not supported
	GDI(x) not supported
	GEI(x) not supported
Si(address)	Similar issues as Ib(). This is read-only and the compiler will issue an error if you attempt to write to this memory using this macro.
SAi(address)	Similar issues as Ib().
SBi(address)	
SCi(address)	
RI(address)	Macros Compatible with 90-70 and 90-30.

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
All(address)	Macros Compatible with 90-70.
AQI(address)	
PI(address)	
LI(address)	
WI(address)	New Macro to support %W memory
Id(address)	Implemented as macro compatible with 90-70 and 90-30 syntax with the exception that the name has been changed from ID to Id. However, this macro does not respect overrides and does not set corresponding transition bits, so the functionality is different than the 90-70 and 90-30. You should use the WritePlcDword() function to get the same functionality as the 90-70 and 90-30. (see next item).
T_INT32 WritePlcDint (T_WORD RefTable, T_DWORD offset, T_DWORD writeValue)	<p>This function writes to reference memory taking into account overrides and transition bits. Reference memory in the specified Reference Table (RefTable) and at the specified “offset” is written with the “writeValue” as a 32-bit signed integer. If the “offset” is out of range, no reference memory values are changed, and the function returns GEF_ERROR. If the offset is within range, the function returns GEF_OK.</p> <p><b>Note:</b> <i>this function will only affect the transition bits that actually change.</i></p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b>  <b>TLIB_ERRNO_READ_ONLY_ER</b>  <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
T_INT32 ReadPlcDint (T_WORD RefTable, T_DWORD offset)	<p>Reference memory in the specified Reference Table (RefTable) and at the specified “offset” is read as a 32-bit signed integer and returned by the function. Errno is set if there is an error reading the value.</p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b>  <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
Qd(address)	Similar issues as Ib()
Md(address)	
Td(address)	
Gd(address)	



Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
	GAD(x) not supported
	GBD(x) not supported
	GCD(x) not supported
	GDD(x) not supported
	GED(x) not supported
Sd(address)	Similar issues as lb(). This is read-only and the compiler will issue an error if you attempt to write to this memory using this macro.
SAd(address)	Similar issues as lb().
SBd(address)	
SCd(address)	
RD(address)	Macros compatible with the 90-70 and 90-30.
AID(address)	
AQD(address)	
PD(address)	Macros compatible with the 90-70.
LD(address)	
WD(address)	New Macro to support %W memory
RF(address)	Macros compatible with the 90-70 and 90-30.
AIF(address)	
AQF(address)	
PF(address)	Macros compatible with the 90-70.
LF(address)	
WF(address)	New Macro to support %W memory
AIDbl(address)	
AQDbl(address)	
LDbl(address)	
PDbl(address)	
RDbl(address)	
WDbl(address)	

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
T_INT32 WritePlcDouble (T_WORD RefTable, T_DWORD offset, T_REAL64 writeValue);	<p>This function writes to reference memory taking into account overrides and transition bits. Reference memory in the specified Reference Table (RefTable) and at the specified “offset” is written with the “writeValue” as a 64-bit floating point value. If the “RefTable” or “offset” are out of range, no reference memory values are changed, and the function returns GEF_ERROR. If the “offset” is within range, the function returns GEF_OK.</p> <p><b>Note:</b> <i>this function will affect only the transition bits that actually change.</i></p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b>  <b>TLIB_ERRNO_READ_ONLY_ER</b>  <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
T_REAL64 ReadPlcDouble (T_WORD RefTable, T_DWORD offset);	<p>Reference memory in the specified Reference Table (RefTable) and at the specified offset is read as a 64-bit floating point value and returned by the function. Errno is set if there is an error reading the value.</p> <p>Errnos:</p> <p><b>TLIB_ERRNO_OFFSET_RANGE_ER</b>  <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b></p>
T_INT32 PlcMemCopy (void *pDestination, void *pSource, T_DWORD size)	<p>This function writes to reference memory taking into account overrides and transition bits. The function writes data pointed to by pDestination based on the memory pointed to by pSource. The length of data written is determined by the “size” parameter which is in units of bytes (8 bits).</p> <p>Errnos:</p> <p><b>TLIB_ERRNO_INVALID_SOURCE_POINTER</b> (Considers pointer and size)  <b>TLIB_ERRNO_INVALID_DEST_POINTER</b> (Considers pointer and size)  <b>TLIB_ERRNO_READ_ONLY_ER</b></p>
BIT_TST_I_TRANS(address)	Macros compatible with the 90-70 and 90-30.
BIT_TST_Q_TRANS(address)	
BIT_TST_M_TRANS(address)	
BIT_TST_T_TRANS(address)	
BIT_TST_G_TRANS(address)	
	BIT_TST_GA_TRANS(address) not supported

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
	BIT_TST_GB_TRANS(address) not supported
	BIT_TST_GC_TRANS(address) not supported
	BIT_TST_GD_TRANS(address) not supported
	BIT_TST_GE_TRANS(address) not supported
BIT_TST_S_TRANS(address)	Macros compatible with 90-70 and 90-30.
BIT_TST_SA_TRANS(address)	
BIT_TST_SB_TRANS(address)	
BIT_TST_SC_TRANS(address)	
IB_TRANS(address)	Macros compatible with 90-70 and 90-30. This is read-only and the compiler will issue an error if you attempt to write to this memory using this macro.
QB_TRANS(address)	
MB_TRANS(address)	
TB_TRANS(address)	
GB_TRANS(address)	
	GAB_TRANS(x) not supported
	GBB_TRANS(x) not supported
	GCB_TRANS(x) not supported
	GDB_TRANS(x) not supported
	GEB_TRANS(x) not supported
SB_TRANS(address)	Macros compatible with 90-70 and 90-30. This is read-only and the compiler will issue an error if you attempt to write to this memory using this macro.
SAB_TRANS(address)	
SBB_TRANS(address)	
SCB_TRANS(address)	
BIT_TST_I_DIAG(address)	Macros compatible with 90-70. This is read-only and the compiler will issue an error if you attempt to write to this memory using this macro.
BIT_TST_Q_DIAG(address)	
IB_DIAG(address)	
QB_DIAG(address)	
AIB_DIAG(address)	
AQB_DIAG(address)	
AI_HIALRM(address)	Macros compatible with 90-70.
AI_LOALRM(address)	

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
AIB_FAULT(address)	Macro compatible with 90-70. AIB_FAULT is non-zero for conditions that set a fault contact or generate a fault entry in the I/O fault table, such as Overrange and Underrange.
AQB_FAULT(address)	Macros compatible with 90-70.
AI_OVERRANGE(address)	
AI_UNDERRANGE(address)	
AQ_OVERRANGE(address)	Macro not supported by 90-70.
AQ_UNDERRANGE(address)	Macro not supported by 90-70.
T_DWORD refMemSize (T_WORD RefTable)	New generic memory size function. The function returns the memory size based on the RefTable segment selector. Errnos: <b>TLIB_ERRNO_INVALID_REF_TABLE_ER</b>
L_SIZE	Compatible with the 90-70 but is implemented as a function; for example: #define L_SIZE refMemSize(L_MEM)
P_SIZE	
R_SIZE	Compatible with the 90-70 and 90-30. Implemented as a function.
AI_SIZE	
AQ_SIZE	
I_SIZE	
Q_SIZE	
T_SIZE	
M_SIZE	
G_SIZE	
	GA_SIZE not supported
	GB_SIZE not supported
	GC_SIZE not supported
	GD_SIZE not supported
	GE_SIZE not supported
SA_SIZE	Compatible with the 90-70 and 90-30 but implemented as a function; for example: <b>#define SA_SIZE refMemSize(SA_MEM)</b>
SB_SIZE	
SC_SIZE	
S_SIZE	

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
W_SIZE	New Macro to support %W memory. Not supported by Series 90.
I_DIAGS_SIZE	Compatible with the 90-70 but implemented as a function; for example: <b>#define I_DIAGS_SIZE refMemSize(I_DIAG_MEM)</b>
Q_DIAGS_SIZE	
AI_DIAGS_SIZE	
AQ_DIAGS_SIZE	
RACKX(r)	Compatible with the 90-70 but implemented as a function call to the function rackX(). Errnos: <b>TLIB_ERRNO_INVALID_RACK</b>
SLOTX(r,s)	Compatible with the 90-70 but implemented as a function call to slotX(); Errnos: <b>TLIB_ERRNO_INVALID_RACK TLIB_ERRNO_INVALID_SLOT</b>
BLOCKX(r,s,b,sba)	Compatible with the 90-70 but implemented as a function call to blockX(); Errnos: <b>REF_ERRNO_INPUT_OUT_OF_RANGE</b>
RSMB(x)	Compatible with the 90-70 but implemented as a function call to rsmb(); Errnos: <b>TLIB_ERRNO_INVALID_RACK</b>
FST_SCN	Macro compatible with 90-70.
LST_SCN	Macro to provide access to the %S00002 (system last scan indication) bit. Compatible with Series 90-30.
T_10MS	Macros compatible with 90-70 and 90-30.
T_100MS	
T_SEC	
T_MIN	
ALW_ON	
ALW_OFF	
SY_FULL	
IO_FULL	

Target Library Reference Memory Functions & Macros	Series 90 PLC Library Compatibility Notes & Issues
FST_EXE	Macro compatible with the 90-70. <i>Note:</i> FST_EXE value is inherited from the calling block.

## A-2 Target Library Fault Table Functions, Structures and Constants

Implemented in ctkPlcFault.h

Target Library Fault Table Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
Fault Table Functions	
<pre> T_INT32 PLCC_read_fault_tables(struct read_fault_tables_rec *x); /* This service request will read the entire PLC or I/O fault table. */  #define PLC_FAULT_TABLE 0 #define IO_FAULT_TABLE 1  #define NUM_LEGACY_PLC_FAULT_ENTRIES 16 #define NUM_LEGACY_IO_FAULT_ENTRIES 32  struct time_stamp_rec{     T_BYTE second; /* BCD format, seconds in low-order nibble, tens of seconds in high-order nibble. */     T_BYTE minute; /* BCD format, same as for seconds. */     T_BYTE hour; /* BCD format, same as for seconds. */     T_BYTE day; /* BCD format, same as for seconds. */     T_BYTE month; /* BCD format, same as for seconds. */     T_BYTE year; /* BCD format, same as for seconds. */ };  struct PLC_flt_address_rec{     T_BYTE rack;     T_BYTE slot;     T_WORD task; };  struct IO_flt_address_rec{     T_BYTE rack;     T_BYTE slot;     T_BYTE IO_bus;     T_BYTE block;     T_WORD point; };  struct reference_address_rec{ </pre>	<p>Compatible with the Series 90 library, with the exception that the union must be named as required by the GNU C compiler. Therefore, to get access to a particular fault, the following syntax must be used:</p> <p>myFaultRec.faultEntry.PLC_faults[0]...</p>

Target Library Fault Table Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_BYTE memory_type; T_WORD offset; };  struct PLC_fault_entry_rec{ T_BYTE long_short; T_BYTE reserved[3]; struct PLCflt_address_rec PLC_fault_address; T_BYTE fault_group; T_BYTE fault_action; T_WORD error_code; T_WORD fault_specific_data[12]; struct time_stamp_rec time_stamp; };  struct IO_fault_entry_rec{ T_BYTE long_short; struct reference_address_rec reference_address; struct IOflt_address_rec IO_fault_address; T_BYTE fault_group; T_BYTE fault_action; T_BYTE fault_category; T_BYTE fault_type; T_BYTE fault_description; T_BYTE fault_specific_data[21]; struct time_stamp_rec time_stamp; };  struct PLC_ext_fault_entry_rec{ T_BYTE long_short; T_BYTE reserved[3]; struct PLCflt_address_rec PLC_fault_address; T_BYTE fault_group; T_BYTE fault_action; T_WORD error_code; T_WORD fault_specific_data[12]; struct ext_time_stamp_rec time_stamp; T_WORD fault_id; };  struct IO_ext_fault_entry_rec{ T_BYTE long_short; struct reference_address_rec reference_address; </pre>	



Target Library Fault Table Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> struct IOflt_address_rec IO_fault_address; T_BYTE fault_group; T_BYTE fault_action; T_BYTE fault_category; T_BYTE fault_type; T_BYTE fault_description; T_BYTE fault_specific_data[21]; struct ext_time_stamp_rec time_stamp; T_WORD fault_id; };  struct read_fault_tables_rec {     T_WORD table; /* PLC_FAULT_TABLE or IO_FAULT_TABLE */     T_WORD zero; /* must be set to zero */     T_WORD reserved[13];     struct time_stamp_rec time_since_clear;     T_WORD num_faults_since_clear;     T_WORD num_faults_in_queue;     T_WORD num_faults_read;     union{         struct PLC_fault_entry_rec PLC_faults[NUM_LEGACY_PLC_FAULT_ENTRIES];         struct IO_fault_entry_rec IO_faults[NUM_LEGACY_IO_FAULT_ENTRIES];     } faultEntry; }; </pre>	

Target Library Fault Table Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_read_last_ext_fault(struct read_last_ext_fault_rec *x); /* Read Last-Logged Extended Fault Table Entry . */  struct PLC_ext_fault_entry_rec{     T_BYTE long_short;     T_BYTE reserved[3];     struct PLC_flt_address_rec PLC_fault_address;     T_BYTE fault_group;     T_BYTE fault_action;     T_WORD error_code;     T_WORD fault_specific_data[12];     struct ext_time_stamp_rec time_stamp; };  struct IO_ext_fault_entry_rec{     T_BYTE long_short;     struct reference_address_rec reference_address;     struct IO_flt_address_rec IO_fault_address;     T_BYTE fault_group;     T_BYTE fault_action;     T_BYTE fault_category;     T_BYTE fault_type;     T_BYTE fault_description;     T_BYTE fault_specific_data[21];     struct ext_time_stamp_rec time_stamp; };  struct read_last_ext_fault_rec {     T_WORD table; /* PLC_EXT_FAULT_TABLE or IO_EXT_FAULT_TABLE */     union {         struct PLC_ext_fault_entry_rec PLC_entry;         struct IO_ext_fault_entry_rec IO_entry;     }; }; #define PLC_EXT_FAULT_TABLE 0x80 #define IO_EXT_FAULT_TABLE 0x81 </pre>	<p>This function is not described in the Series 90 C Toolkit User's Manual but is included in the 90-70/90-30 C Toolkit header files. This function is included in the PACSystems C Toolkit for compatibility.</p>

<pre> T_INT32 PLCC_read_last_fault(struct read_last_fault_rec *x); /* Read Last-Logged Fault Table Entry. */ struct time_stamp_rec{     T_BYTE second; /* BCD format, seconds in low-order nibble, */         /* tens of seconds in high-order nibble. */     T_BYTE minute; /* BCD format, same as for seconds. */     T_BYTE hour; /* BCD format, same as for seconds. */     T_BYTE day; /* BCD format, same as for seconds. */     T_BYTE month; /* BCD format, same as for seconds. */     T_BYTE year; /* BCD format, same as for seconds. */ }; struct PLCflt_address_rec{     T_BYTE rack;     T_BYTE slot;     T_WORD task; };  struct IOflt_address_rec{     T_BYTE rack;     T_BYTE slot;     T_BYTE IO_bus;     T_BYTE block;     T_WORD point; }; struct PLC_fault_entry_rec{     T_BYTE long_short;     T_BYTE reserved[3];     struct PLCflt_address_rec PLC_fault_address;     T_BYTE fault_group;     T_BYTE fault_action;     T_WORD error_code;     T_WORD fault_specific_data[12];     struct time_stamp_rec time_stamp; };  struct IO_fault_entry_rec{     T_BYTE long_short;     struct reference_address_rec reference_address;     struct IOflt_address_rec IO_fault_address;     T_BYTE fault_group;     T_BYTE fault_action; </pre>	<p>Compatible with 90-70 and 90-30.</p>
---	---

Target Library Fault Table Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_BYTE fault_category; T_BYTE fault_type; T_BYTE fault_description; T_BYTE fault_specific_data[21]; struct time_stamp_rec time_stamp; T_WORD fault_id; }; struct read_last_fault_rec {     T_WORD table; /* PLC_FAULT_TABLE or IO_FAULT_TABLE */     union {         struct PLC_fault_entry_rec PLC_entry;         struct IO_fault_entry_rec IO_entry;     }; }; </pre>	
<pre> T_INT32 PLCC_clear_fault_tables(struct clear_fault_tables_rec *x); /* Clear Fault Tables */ struct clear_fault_tables_rec{     T_WORD table; }; #define PLC_FAULT_TABLE 0 #define IO_FAULT_TABLE 1 </pre>	Compatible with 90-70 and 90-30.

<pre> T_INT32 PLCC_read_ext_fault_tables (struct read_ext_fault_tables_rec *x); /* Read Extended Fault Tables */  struct read_ext_fault_tables_rec {     T_WORD table; /* PLC_EXT_FAULT_TABLE or IO_EXT_FAULT_TABLE */     T_WORD start_index;     T_WORD number_of_entries_to_read;     T_WORD reserved[12];     struct time_stamp_rec time_since_clear;     T_WORD num_faults_since_clear;     T_WORD num_faults_in_queue;     T_WORD num_faults_read;     T_WORD PlcName[16]; union{         struct PLC_ext_fault_entry_rec PLC_faults[1];         struct IO_ext_fault_entry_rec IO_faults[1];     } faultEntry; }; Example: Extended fault table structure declaration with user defined number of fault entries: /* Constants / #defines */ #define MY_PLC_FLT_TBL_SIZE 64 #define MY_IO_FLT_TBL_SIZE 64 /* Structures and typedefs */ /* Note: this structure must be packed */ #pragma pack(1) struct my_read_ext_fault_tables_rec {     T_WORD table; /* PLC_EXT_FAULT_TABLE or IO_EXT_FAULT_TABLE */     T_WORD start_index;     T_WORD number_of_entries_to_read;     T_WORD reserved[12];     struct time_stamp_rec time_since_clear;     T_WORD num_faults_since_clear;     T_WORD num_faults_in_queue;     T_WORD num_faults_read;     T_WORD PlcName[16];     union     {         struct PLC_ext_fault_entry_rec PLC_faults[MY_PLC_FLT_TBL_SIZE];         struct IO_ext_fault_entry_rec IO_faults[MY_IO_FLT_TBL_SIZE];     } faultEntry; }; </pre>	<p>This function is not described in the Series 90 C Toolkit User's Manual but is included in the 90-70 C Toolkit header file. This is included in the PACSystems C Toolkit for compatibility. This function maps to service request 20 in the PACSystems. Since the size of the extended fault table can be variable Depending on the model of the PACSystems CPU, you will need to create your own structure with the same members and dimension PLC_faults and IO_faults members to the size of the maximum number of faults you want to read. You must then declare a variable of this type and cast it to a read_ext_fault_tables_rec when calling this function (See Example – Note that the structure must be packed to work properly)</p> <p>Another issue is that the union must be named as required by the GNU C compiler. Therefore, to get access to a particular fault, the following syntax must be used:</p> <p>myExtFaultRec.faultEntry.PLC_faults[0]...</p>
--	---

Target Library Fault Table Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre>#pragma pack() /* Variable Declaration and Calling Example */ struct my_read_ext_fault_tables_rec readExtFaultTablesRec; PLCC_read_ext_fault_tables(     (struct read_ext_fault_tables_rec*)&amp;readExtFaultTablesRec);</pre>	

## A-3 Target Library General Functions, Structures and Constants

Implemented in ctkPlcFunc.h

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<b>General PLC Functions</b>	
<pre>T_INT32 PLCC_read_elapsed_clock (struct elapsed_clock_rec *); struct elapsed_clock_rec {     T_DWORD seconds;     T_WORD hundred_usec; };</pre>	Compatible with 90-70 and 90-30 libraries.
<pre>T_INT32 PLCC_read_nano_elapsed_clock (struct nano_elapsed_clock_rec *); struct nano_elapsed_clock_rec {     T_DWORD seconds;     T_DWORD nanoseconds; };</pre>	Function returns elapsed time in nanoseconds.
<pre>T_INT32 PLCC_chars_in_printf_q (void); /* integer value equal to number of characters currently in the printf buffer */  #define PRINTF_Q_SIZE 2048</pre>	<p>Returns GEF_NOT_SUPPORTED since printf is not supported. The following functions provide information on the number of characters in the input/output queues:</p> <p>PLCC_CharsInMessageWriteQ PLCC_CharsInMessageReadQ</p>
<pre>T_INT32 PLCC_gen_alarm ( word, char * ); /* Log a user specified application fault in the PLC fault table. */</pre>	Compatible with the 90-70 and 90-30 libraries.

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_get_plc_version ( struct PLC_ver_info_rec * ); /* Get the PLC family, model, and firmware version and revision. */ struct PLC_ver_info_rec {     T_WORD    family;    /* Host PLC product line */     T_WORD    model;    /* Specific Model of PLC */     T_BYTE    sw_ver;    /* Major Version of PLC firmware */     T_BYTE    sw_rev;    /* Minor Revision of PLC firmware */ };  #define FAMILY_PACSYSTEMS 0x2002 </pre>	<p>Compatible with 90-70 and 90-30 libraries.</p>
<pre> T_INT32 PLCC_comm_req(struct comm_req_rec *x); /* Communications Request */ struct status_addr{     T_WORD seg_selector;     T_WORD offset; };  struct comm_req_command_blk_rec{     T_WORD length;     T_WORD wait;     struct status_addr status;     T_WORD idle_timeout;     T_WORD max_comm_time;     T_WORD data[128]; };  struct comm_req_rec{     struct comm_req_command_blk_rec *command_blk;     T_BYTE slot;     T_BYTE rack;     T_DWORD task_id; }; </pre>	<p>Compatible with the 90-70 and 90-30 but not able to access full range of %W memory. Use PLCC_comm_req_extended() to provide access to the full %W address range.</p>

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_comm_req_extended (struct comm_req_rec *x); /* Communications Request */ struct status_addr_extended{     T_WORD seg_selector;     T_DWORD offset; };  struct comm_req_command_blk_rec{     T_WORD length;     T_WORD wait;     struct status_addr_extended status;     T_WORD idle_timeout;     T_WORD max_comm_time;     T_WORD data[128]; };  struct comm_req_rec_extended{     struct comm_req_command_blk_rec_extended *command_blk;     T_BYTE slot;     T_BYTE rack;     T_DWORD task_id; } </pre>	<p>Has the same functionality as PLCC_comm._req, except that it can access the full address range of %W memory. Not supported by Series 90 PLCs.</p>
<pre> T_INT32 PLCC_do_io(struct do_io_rec *x); /* Do I/O */ struct do_io_rec{     T_BYTE start_mem_type;     T_WORD start_mem_offset;     T_WORD length;     T_BYTE alt_mem_type; /* must be set to NULL_SEGSEL if not used */     T_WORD alt_mem_offset; }; </pre>	<p>Compatible with 90-70 and 90-30 libraries. Errnos: <b>TLIB_ERRNO_DOIO_INVALID_IO_REF_ER</b> <b>TLIB_ERRNO_DOIO_INVALID_AUX_REF_ER</b></p>



Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_do_io_ext(struct do_io_ext_rec *x);  struct do_io_ext_rec{   T_WORD start_mem_type;   T_DWORD start_mem_offset;   T_DWORD length; /* Ignored if start_mem_type is PLC_VAR_MEM */   T_WORD alt_mem_type; /* must be set to NULL_SEGSEL if not used */   T_DWORD alt_mem_offset; }; </pre>	<p>Not supported by Series 90. Supported by PACSystems Release 3.5 or greater.</p>
	<p>The Enhanced DO_IO function (Series 90-30 only) is not supported.</p>
<pre> T_INT32 PLCC_sus_io(void); /* Suspend I/O */ </pre>	<p>Compatible with the 90-70 and 90-30 library.</p>
<pre> T_INT32 PLCC_scan_set_io(struct scan_set_io_rec *pScanSetIoRec);  struct scan_set_io_rec{   T_BOOLEAN scan_inputs;   T_BOOLEAN scan_outputs;   T_UINT16 scan_set_number; }; </pre>	<p>Not supported by Series 90. Supported by PACSystems Release 5.0 or greater.</p>
<pre> T_INT32 PLCC_SNP_ID(T_BYTE request_type, char *id_str_ptr); /* Read or Write SNP ID */ #define READ_ID 0 #define WRITE_ID 1 </pre>	<p>Compatible with 90-70 and 90-30 libraries (Release 2.0 and later).</p>

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_read_override(T_BYTE tbl_typ, T_WORD ref_num, T_WORD len, T_BYTE *data); /* Error return values */ #define BAD_MEMORY_TYPE -2 #define OFFSET_NOT_BYTE_ALIGNED -3 #define READING_OUTSIDE_REF_MEM -4 #define BAD_DATA_POINTER -5  /* Read Overrides */ #define I_OVR 0 #define Q_OVR 1 #define M_OVR 2 #define G_OVR 3 </pre>	<p>The following 90-70 and 90-30 values are not supported by PACSystems:</p> <pre> #define GA_OVR 4 #define GB_OVR 5 #define GC_OVR 6 #define GD_OVR 7 #define GE_OVR 8 </pre>
	<pre> int far PLCCinvokeldblock(void); (not supported) </pre>
<pre> T_INT32 PLCC_MessageWrite(T_INT32 port, char *buffer, T_INT32 numBytes);  #define PORT1 0 #define PORT2 1 </pre>	<p>New function to provide serial output. Note: for all PLCC_Message* functions, the Hardware configuration for the serial port must be setup for Message Mode for the function to access the serial port.</p> <p>Errnos:</p> <pre> TLIB_ERRNO_MSG_INVALID_PORT TLIB_ERRNO_MSG_NOT_CONFIGURED TLIB_ERRNO_MSG_INVALID_LENGTH </pre>
<pre> T_INT32 PLCC_MessageRead(T_INT32 port, char *buffer, T_INT32 numBytes); </pre>	<p>New function to provide serial input.</p> <p>Errnos:</p> <pre> TLIB_ERRNO_MSG_INVALID_PORT TLIB_ERRNO_MSG_NOT_CONFIGURED TLIB_ERRNO_MSG_INVALID_LENGTH </pre>
<pre> T_INT32 PLCC_CharsInMessageWriteQ(T_INT32 port); </pre>	<p>New function that Returns the number of bytes in the write queue.</p> <p>Errnos:</p> <pre> TLIB_ERRNO_MSG_INVALID_PORT TLIB_ERRNO_MSG_NOT_CONFIGURED </pre>
<pre> T_INT32 PLCC_CharsInMessageRead(T_INT32 port, char *buffer, T_INT32 numBytes); </pre>	<p>New function that returns the number of bytes in the read queue</p> <p>Errnos:</p> <pre> TLIB_ERRNO_MSG_INVALID_PORT TLIB_ERRNO_MSG_NOT_CONFIGURED </pre>
Functions based on service requests from the SVCREQ function block	

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_const_sweep_timer(struct const_sweep_timer_rec *x); /* Change/Read Constant Sweep Timer. */ struct const_sweep_input_rec {     T_WORD action;     T_WORD timer_value; };  /* structure with return values */ struct const_sweep_output_rec {     T_WORD sweep_mode;     T_WORD current_time_value; };  struct const_sweep_timer_rec {     union {         struct const_sweep_input_rec input;         struct const_sweep_output_rec output;     }; };  /* action values */ #define DISABLE_CONSTANT_SWEEP_MODE 0 #define ENABLE_CONSTANT_SWEEP_MODE 1 #define CHANGE_TIMER_VALUE 2 #define READ_TIMER_VALUE_AND_STATE 3  /* sweep mode return values */ #define CONSTANT_SWEEP_ENABLED 1 #define CONSTANT_SWEEP_DISABLED 0 </pre>	<p>Compatible with 90-70 and 90-30 except that Microcycle (90-70) is not supported.</p>

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_read_window_values(struct read_window_values_rec *x); /* Read Window Values. */ /* window modes */ #define LIMITED_MODE      0 #define CONSTANT_MODE     1 #define RUN_TO_COMPLETION_MODE 2  /* structure with return values */ struct read_window_values_rec{     T_BYTE controller_win_time;     T_BYTE controller_win_mode; /* LIMITED_MODE, CONSTANT_MODE, */                                 /* RUN_TO_COMPLETION_MODE */     T_BYTE backplane_comm_win_time;     T_BYTE backplane_comm_win_mode; /* LIMITED_MODE, CONSTANT_MODE, */                                 /* RUN_TO_COMPLETION_MODE */     T_BYTE background_win_time;     T_BYTE background_win_mode; /* LIMITED_MODE, CONSTANT_MODE, */                                 /* RUN_TO_COMPLETION_MODE */ }; </pre>	<p>Compatible with the 90-70 and 90-30 except that structure member names with the “prog_” suffix now use the “controller_” suffix and those that use the “sys_” suffix now use the “backplane_” suffix. This is to make the names consistent with the PACSystems terminology.</p>
<pre> T_INT32 PLCC_change_controller_comm_window(struct change_controller_comm_window_rec *x); /* Change Controller Communications Window State and Values */ /* input structure */ struct change_controller_comm_window_rec{     T_BYTE time;     T_BYTE mode; /* LIMITED_MODE, CONSTANT_MODE, */                 /* RUN_TO_COMPLETION_MODE */ }; /* window modes */ #define LIMITED_MODE      0 #define CONSTANT_MODE     1 #define RUN_TO_COMPLETION_MODE 2 </pre>	<p>Compatible with the 90-70 and 90-30, except that function and structure names containing “prog_” now use the “controller_”. This makes the names consistent with the PACSystems terminology.</p>

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_change_backplane_comm_window (struct change_backplane_comm_window_rec *x); /* Change Backplane Communications Window State and Values */ struct change_system_comm_window_rec{     T_BYTE time;     T_BYTE mode; /* LIMITED_MODE, CONSTANT_MODE, */                 /* RUN_TO_COMPLETION_MODE */ }; </pre>	<p>Compatible with the 90-70 and 90-30, except that function and structure names containing “system_” now use the “backplane_” This makes the names consistent with the PACSystems terminology.</p>
<pre> T_INT32 PLCC_change_background_window (struct change_background_window_rec *x); /* Change Background Window State and Values. */ struct change_background_window_rec{     T_BYTE time;     T_BYTE mode; /* LIMITED_MODE, CONSTANT_MODE, */                 /* RUN_TO_COMPLETION_MODE */ }; </pre>	<p>Compatible with 90-70 and 90-30 libraries.</p>
<pre> T_INT32 PLCC_number_of_words_in_chksm(struct number_of_words_in_chksm_rec *x); /* Set/Read Number of Words to Checksummed */ struct number_of_words_in_chksm_rec{     T_WORD read_set;     T_WORD word_count; /* number of words checksummed */ };  #define READ_CHECKSUM_WORDS 0 #define SET_CHECKSUM_WORDS 1 </pre>	<p>Compatible with 90-70 and 90-30 libraries.</p>

<pre> T_INT32 PLCC_tod_clock(struct tod_clock_rec *x); /*Change/Read Time-of-Day Clock State and Values */ #define NUMERIC_DATA_FORMAT 0 #define BCD_FORMAT 1 #define UNPACKED_BCD_FORMAT 2 #define PACKED_ASCII_FORMAT 3 #define POSIX_FORMAT 4 #define NUMERIC_DATA_FORMAT_4_DIG_YR 0x80 #define BCD_FORMAT_4_DIG_YR 0x81 #define UNPACKED_BCD_FORMAT_4_DIG_YR 0x82 #define PACKED_ASCII_FORMAT_4_DIG_YR 0x83 #define SUNDAY 1 #define MONDAY 2 #define TUESDAY 3 #define WEDNESDAY 4 #define THURSDAY 5 #define FRIDAY 6 #define SATURDAY 7  struct num_tod_rec{ T_WORD year; T_WORD month; T_WORD day_of_month; T_WORD hours; T_WORD minutes; T_WORD seconds; T_WORD day_of_week; }; struct BCD_tod_rec{ T_BYTE year; T_BYTE month; T_BYTE day_of_month; T_BYTE hours; T_BYTE minutes; T_BYTE seconds; T_BYTE day_of_week; T_BYTE null; }; struct BCD_tod_4_rec{ T_BYTE year_lo; T_BYTE year_hi; </pre>	<p>Compatible with 90-70 and 90-30 libraries. Some additional formats at available on PACSystems such as BCD_tod_4_rec, unpacked_bcd_tod_4_rec and ascii_tod_4_rec.</p>
---	---

```

T_BYTE month;
T_BYTE day_of_month;
T_BYTE hours;
T_BYTE minutes;
T_BYTE seconds;
T_BYTE day_of_week;
};

struct unpacked_BCD_rec{
    T_BYTE yearlo;
    T_BYTE yearhi;
    T_BYTE monthlo;
    T_BYTE monthhi;
    T_BYTE day_of_month_lo;
    T_BYTE day_of_month_hi;
    T_BYTE hourslo;
    T_BYTE hourshi;
    T_BYTE minslo;
    T_BYTE minshi;
    T_BYTE secslo;
    T_BYTE secshi;
    T_WORD day_of_week;
};

struct unpacked_bcd_tod_4_rec{
    T_WORD hunst_year;
    T_WORD tens_year;
    T_WORD month;
    T_WORD day_of_month;
    T_WORD hours;
    T_WORD minutes;
    T_WORD seconds;
    T_WORD day_of_week;
};

struct ASCII_tod_rec{
    T_BYTE yearhi;
    T_BYTE yearlo;
    T_BYTE space1;
    T_BYTE monthhi;
    T_BYTE monthlo;
    T_BYTE space2;
    T_BYTE day_of_month_hi;
    T_BYTE day_of_month_lo;
    T_BYTE space3;

```

<pre> T_BYTE hourshi; T_BYTE hourslo; T_BYTE colon1; T_BYTE minshi; T_BYTE minslo; T_BYTE colon2; T_BYTE secshi; T_BYTE secslo; T_BYTE space4; T_BYTE day_of_week_hi; T_BYTE day_of_week_lo; }; struct ascii_tod_4_rec{     T_BYTE hun_year_hi;     T_BYTE hun_year_lo;     T_BYTE year_hi;     T_BYTE year_lo;     T_BYTE space1;     T_BYTE month_hi;     T_BYTE month_lo;     T_BYTE space2;     T_BYTE day_of_month_hi;     T_BYTE day_of_month_lo;     T_BYTE space3;     T_BYTE hours_hi;     T_BYTE hours_lo;     T_BYTE colon1;     T_BYTE minutes_hi;     T_BYTE minutes_lo;     T_BYTE colon2;     T_BYTE seconds_hi;     T_BYTE seconds_lo;     T_BYTE space4;     T_BYTE day_of_week_hi;     T_BYTE day_of_week_lo; };  #define READ_CLOCK 0 #define WRITE_CLOCK 1  typedef long int time_t;  struct timespec {     time_t tv_sec;     long int tv_nsec; </pre>	<p>The union must be named as required by the GNU C compiler. Therefore, to get access to a particular member in the "record" union of the tod_clock_rec structure, the following syntax must be used:</p> <pre> todClockRec.record.num_tod.seconds </pre>
---	--



Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> ];  struct tod_clock_rec{     T_WORD read_write; /* READ_CLOCK or WRITE_CLOCK */     T_WORD format; /* NUMERIC_DATA_FORMAT, BCD_FORMAT */         /* UNPACKED_BCD_FORMAT, PACKED_ASCII_FORMAT */     union {         struct num_tod_rec num_tod;         struct BCD_tod_rec BCD_tod;         struct BCD_tod_4_rec BCD_tod_4;         struct unpacked_BCD_rec unpacked_BCD_tod;         struct unpacked_bcd_tod_4_rec unpacked_BCD_tod_4;         struct ASCII_tod_rec ASCII_tod;         struct ascii_tod_4_rec ASCII_tod_4;         struct timespec POSIX_tod;     }; }; </pre>	
	T_INT32 PLCC_tod_clock_with_status(struct tod_clock_with_status_rec *x);Not Supported.
<pre> T_INT32 PLCC_reset_watchdog_timer(void); /* Reset Watchdog Timer */ </pre>	Compatible with 90-70 and 90-30 libraries.
<pre> T_int32 PLCC_time_since_start_of_sweep(struct time_since_start_of_sweep_rec *x); /* Read Sweep Time from the Beginning of Sweep */ struct time_since_start_of_sweep_rec{     T_WORD time_since_start_of_sweep; }; </pre>	Compatible with 90-70 and 90-30 libraries.
<pre> T_INT32 PLCC_nano_time_since_start_of_sweep(struct nano_time_since_start_of_sweep_rec *x); /* Read Sweep Time in nanoseconds from the Beginning of Sweep */ struct nano_time_since_start_of_sweep_rec{     T_DWORD time_since_start_of_sweep; }; </pre>	New function. Provides time in nanosecond units.

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre>T_INT32 PLCC_read_folder_name(struct read_folder_name_rec *x); /* Read Folder Name */ struct read_folder_name_rec{     char folder_name[32]; /* NULL terminated */ };</pre>	Change in number of characters in name to 32 including the NULL terminator.
<pre>T_INT32 PLCC_read_PLC_ID(struct read_PLC_ID_rec *x); /* Read PLC ID */ struct read_PLC_ID_rec{     char PLC_ID[8]; /* NULL terminated */ };</pre>	Compatible with 90-70 and 90-30.
<pre>T_INT32 PLCC_read_PLC_state(struct read_PLC_state_rec *x); /* Read PLC Run State */ #define RUN_DISABLED 1 #define RUN_ENABLED 2  struct read_PLC_state_rec{     T_WORD state; };</pre>	Compatible with 90-70 and 90-30.
<pre>T_INT32 PLCC_shut_down_plc(T_WORD numberOfSweeps); /* Shut Down PLC */</pre>	Compatible with the 90-70 and 90-30 except the function takes an input parameter, number of sweeps, that indicates the number of full sweeps to execute after the function is called.
<pre>T_INT32 PLCC_mask_IO_interrupts(struct mask_IO_interrupts_rec *x); /* Mask/Unmask I/O Interrupt */ struct mask_IO_interrupts_rec{     T_WORD mask; /* MASK or UNMASK */     T_WORD memory_type;     T_WORD memory_address; }; #define MASK 1 #define UNMASK 0</pre>	Compatible with 90-70 and 90-30.

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_mask_IO_interrupts_ext (struct mask_IO_interrupts_ext_rec *x); struct mask_IO_interrupts_ext_rec{     T_WORD action;    /* MASK or UNMASK */     T_WORD memory_type; /* Address of input interrupt trigger */     T_DWORD memory_offset; }; </pre>	<p>Not supported by Series 90. Supported by PACSystems Release 3.5 or greater.</p>
<pre> T_INT32 PLCC_read_IO_override_status(struct read_IO_override_status_rec *x); /* Read I/O Override Status */ struct read_IO_override_status_rec{     T_WORD override_set; };  #define OVERRIDES_SET      1 #define NO_OVERRIDES_SET  0 </pre>	<p>Compatible with 90-70 and 90-30.</p>
<pre> T_INT32 PLCC_set_run_enable(struct set_run_enable_rec *x); /* Set Run Enable/Disable */  #define RUN_DISABLED  1 #define RUN_ENABLED   2  struct set_run_enable_rec{     T_WORD enable; }; </pre>	<p>Compatible with 90-70.</p>

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre> T_INT32 PLCC_mask_timed_interrupts(struct mask_timed_interrupts_rec *x); /* Mask/Unmask Timed Interrupts */  struct mask_timed_interrupts_rec{     T_WORD action; /* READ_INTERRUPT_MASK or WRITE_INTERRUPT_MASK */     T_WORD status; /* if action is READ_INTERRUPT_MASK then this */                 /* field has MASK or UNMASK as the return value */                 /* if the action is WRITE_INTERRUPT_MASK then */                 /* set this field to MASK or UNMASK */ };  ; #define READ_INTERRUPT_MASK 0 #define WRITE_INTERRUPT_MASK 1 </pre>	Compatible with 90-70 and 90-30.
<pre> T_INT32 PLCC_sus_res_HSC_interrupts(struct sus_res_HSC_interrupts_rec *x); /* Suspend/Resume High Speed Counter Interrupts */ #define SUSPEND 1 #define RESUME 0 #define I_BIT 70 #define AI_MEM 10  struct sus_res_HSC_interrupts_rec{     T_WORD action; /* SUSPEND or RESUME */     T_WORD memory_type;     T_WORD reference_address; }; </pre>	Compatible with 90-70 and 90-30.
<pre> T_INT32 PLCC_sus_res_interrupts_ext(struct sus_res_interrupts_ext_rec *x);  struct sus_res_interrupts_ext_rec{     T_WORD action; /* SUSPEND or RESUME */     T_WORD memory_type; /* Address of the interrupt trigger */     T_DWORD memory_offset; }; </pre>	Not supported by Series 90. Supported by PACSystems Release 3.5 or greater.

Target Library General Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
	<p>int PLCC_acc_mem (struct plcc_mem_acc_rec *mem_acc_rec_ptr);</p> <p>Not Supported since bulk memory is supported directly through %W memory type.</p>
<p>T_INT32 PLCC_get_escm_status (struct escm_status_rec *);</p> <p>/* Function PLCC_get_escm_status */</p> <p>struct escm_status_rec {</p> <p>    T_WORD port_number;</p> <p>    T_WORD port_status;</p> <p>};</p>	<p>Compatible with 90-70 except the function will always return 0 (escm not available or unsupported) for this release of PACSystems because the ESCM is not present.</p>
<p>T_INT32</p> <p>PLCC_set_application_redundancy_mode(T_WORD mode);</p> <p>/* Possible values for the backup mode. */</p> <p>#define BACKUP_MODE 0</p> <p>#define ACTIVE_MODE 1</p>	<p>Not supported by Series 90.</p> <p>Supported by PACSystems Release 5.0 or greater.</p>

## A-4 Target Library VME Functions, Structures and Constants

Implemented in ctkPlcBus.h – Compatible with Rx7 only

Target Library VME Functions, Structures and Constants	90-70 PLC Library Compatibility Notes & Issues
	byte PLCC_VME_set_amcode(byte amcode) function is not supported since the PACSystems system uses rack, slot, sub-slot, region to address VME memory.
T_INT32 PLCC_BUS_read_byte(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_BYTE *value, T_DWORD address); /* Read a byte from the VME bus. */	Similar function as the 90-70 but the function now has four additional input parameters, rack, slot, sub-slot and region, that specify the VME memory access. In addition, the functions now have a status parameter and the name uses “BUS” instead of “VME” to make the function more general (i.e. the same code could be used on various PACSystems CPUs)
T_INT32 PLCC_BUS_read_word(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_WORD *value, T_DWORD address); /* Read a word from the VME bus. */	
T_INT32 PLCC_BUS_read_block(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, void *buffer, T_WORD length, T_DWORD address); /* Read a block from the VME bus */	
T_INT32 PLCC_BUS_write_byte(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_BYTE value, T_DWORD address); /* Write a byte to the VME bus */	
T_INT32 PLCC_BUS_write_word(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_WORD value, T_DWORD address); /* Write a word to the VME bus. */	
T_INT32 PLCC_BUS_write_block(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, void *buffer, T_WORD length, T_DWORD address); /* Write a block of data to the VME bus */	
	word PLCC_VME_config_read(void *buffer, word length, byte rack, byte slot, unsigned long offset); Not supported.

Target Library VME Functions, Structures and Constants	90-70 PLC Library Compatibility Notes & Issues
	word PLCC_VME_config_write(void *buffer, word length, byte rack, byte slot, unsigned long offset); Not supported.
<p>T_INT32 PLCC_BUS_RMW_byte (T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_DWORD *pOriginalValue, T_BYTE op_type, T_BYTE mask, T_DWORD address);</p> <p>/* Read Modify Write a byte to the VME bus */</p> <p>#define BUS_OR 1</p> <p>#define BUS_AND 0</p>	<p>Similar function as the 90-70 but the function now has four additional input parameters, rack, slot, sub-slot and region, that specify the VME memory access. In addition, the functions now have a status parameter and the name uses “BUS” instead of “VME” to make the function more general (i.e. the same code could be used on various PACSystems CPUs)</p>
<p>T_INT32 PLCC_BUS_RMW_word (T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_DWORD *pOriginalValue, T_BYTE op_type, T_WORD mask, T_DWORD address);</p> <p>/* Read Modify Write a word to the VME bus */</p> <p>#define BUS_OR 1</p> <p>#define BUS_AND 0</p>	
<p>T_INT32 PLCC_BUS_TST_byte (T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_BYTE *semaphore_output, T_DWORD address);</p> <p>/* Test and set a byte on the VME bus */</p>	
<p>T_INT32 PLCC_BUS_TST_word (T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_WORD *semaphore_output, T_DWORD address);</p> <p>/* Test and set a word on the VME bus */</p>	
<p>T_INT32 PLCC_BUS_read_dword(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_DWORD *value, T_DWORD address);</p> <p>/* Read a dword from the VME bus. */</p>	New Bus function for 32-bit access
<p>T_INT32 PLCC_BUS_write_dword(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_DWORD value, T_DWORD address);</p> <p>/* Write a dword to the VME bus */</p>	New Bus function for 32-bit access

Target Library VME Functions, Structures and Constants	90-70 PLC Library Compatibility Notes & Issues
<pre>T_INT32 PLCC_BUS_RMW_dword(T_WORD rack, T_WORD slot, T_WORD subSlot, T_WORD region, T_WORD *pStatus, T_DWORD *pOriginalValue, T_BYTE op_type, T_DWORD mask, T_DWORD address); /* Read Modify Write a dword to the VME bus */ #define BUS_OR 1 #define BUS_AND 0</pre>	<p>New Bus function for 32-bit access</p>

## A-5 Target Library Error Functions, Structures and Constants

Implemented in ctkPlcErno.h

Target Library Error Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<pre>void PLCC_ClearErno(void);</pre>	<p>This is a new function. It clears the errno in the current context. As a general rule, this function should be called just before calling a function whose status will be checked by using PLCC_GetErno. If this is not done, the Erno value could be the result of previous function call.</p>
<pre>int PLCC_GetErno(void)</pre>	<p>This is a new function. It returns the errno in the current context. errno contains the error code set by the last Target Library or C Run Time Library function to declare an error.</p>

## A-6 Target Library Utility Functions, Structures and Constants

Implemented in ctkPlcUtil.h



Target Library Utility Functions, Structures and Constants	Series 90 PLC Library Compatibility Notes & Issues
<p>T_WORD PLCC_Crc16Checksum (T_BYTE *pFirstByte, T_DWORD length, T_WORD currentCrcValue);</p>	<p>This is a new function. It calculates a CRC16 checksum over the given area with the given starting value and length in bytes. The currentCrcValue is normally 0. When checking a large memory range section by section, one can use the previous section's CRC value as the initial value.</p> <p>Errnos: TLIB_ERRNO_UTIL_NULL_POINTER</p>

## Appendix B: C Run-Time Library Functions

The library functions listed in this appendix do not set errno, unless otherwise indicated.

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <stdio.h>	<u>Input/Output:</u>	<p>The Series 90-70 function, printf() is not supported on the target and will return GEF_ERROR. The following lines provide equivalent printf functionality:</p> <pre>char buffer[100]; T_INT32 numBytes;  numBytes=sprintf(buffer, "my Message\r\n"); PLCC_MessageWrite(PORT1, buffer, numBytes);</pre> <p>When debugging on the PC, printf is supported or you can use the sprintf/ PLCC_MessageWrite combination shown above.</p>
#include <stdio.h>	int sprintf(char*, const char* format, ...);	
#include <stdio.h>	int sscanf (const char* string, const char* format, ...);	New function to PACSystems; i.e. it was not supported on Series 90 PLCs.
#include <math.h>	<u>Math:</u>	<p>Note for the following Math functions:</p> <p>+ -NAN is 0x7ff8000000000000 &amp; 0xfff8000000000000 respectively for a double value.</p> <p>+ -Infinity is 0x7ff0000000000000 and 0xfff0000000000000 respectively for a double value.</p> <p>+ -NAN is 0x7f8xxxxx and 0xff8xxxxx respectively where xxxxx is non-zero for a float value.</p> <p>+ -Infinity is 0x7f800000 and 0xff800000 respectively for a float value.</p>
#include <math.h>	double acos(double); (64-bit), float acosf(float); (32-bit)	<p>acos() (32-bit) on the 90-70 is functionally equivalent to acosf() on PACSystems. acosl() (80 bits) is not supported.</p> <p>Similar compatibility issues exist for the other math functions.</p> <p>Errno exception: EDOM is not set by this function and returns "not a number" + -NAN if outside the range of -1 to 1</p>

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <math.h>	double asin(double), float asinf(float);	asinl() is not supported Errno exception: EDOM is not set by this function and returns "not a number" +-NAN if outside the range of -1 to 1
#include <math.h>	double atan(double), float atanf(float);	atanl() is not supported. Function does not set errno. For +-NAN input returns +-NAN respectively.
NA		_cabs() is not supported.
NA		_cabsf() is not supported.
#include <math.h>	double ceil(double), float ceilf(float);	ceil() is not supported Function does not set errno. For +-NAN input returns +-NAN respectively. For +-Infinity input returns +-Infinity respectively.
#include <math.h>	double cos(double), float cosf(float);	cosl() is not supported. Function does not set errno. For +-NAN input returns +-NAN respectively. For +-Infinity input returns -NAN.
#include <math.h>	double cosh(double), float coshf(float);	coshl() is not supported. Function does not set errno. For +-NAN input returns +-NAN respectively. For +-Infinity input returns +NAN.
#include <math.h>	double exp(double), float expf(float);	expl() is not supported. Errno exceptions: ERANGE or EDOM are not set by this function and the functions returns +NAN when the input is +Infinity or +-NAN. The function returns +Infinity if the input is -Infinity.
#include <math.h>	double fabs(double), float fabsf(float);	fabsl() is not supported. Errno & return exceptions: EDOM and ERANGE are not set. A +- Infinity input returns a +Infinity value. A +- NAN input returns a +NAN value.
#include <math.h>	double floor(double), float floorf(float);	floorl() is not supported. Function does not set errno. For +-NAN input returns +-NAN respectively. For +-Infinity input returns +-Infinity respectively.
#include <math.h>	double fmod(double x, double y), float fmodf(float x, float y);	fmodl() is not supported. Errno & return value exceptions: EDOM is not set. If y = 0, the return value is +NAN.

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <math.h>	double frexp(double x, int *y);	frexpl() is not supported Errno: sets EDOM for x = +-NAN or +-Infinity.
NA		_hypot() is not supported .(calculates the hypotenuse).
NA		_hypotl is not supported.
#include <math.h>	double ldexp(double x, int y);	ldexpl is not supported. Errno: set errno to EDOM for x +-NAN and ERANGE for x +-Infinity. Caution: setting y > 65535 could cause the PLC watchdog to time out.
#include <math.h>	double log(double x), float logf(float x);	logl() is not supported. Errno and return exceptions: EDOM is not set for a negative input. ERANGE is not set for an input of 0. x < 0 returns -NAN x=+Infinity returns +Infinity x=0 returns -Infinity x=+-NAN returns +-NAN respectively.
#include <math.h>	double log10(double x), float log10f(float x);	log10l() is not supported Errno and return exceptions: EDOM is not set for a negative input. ERANGE is not set for an input of 0. x < 0 returns -NAN x=+Infinity returns +Infinity x=0 returns -Infinity x=+-NAN returns +-NAN respectively.
#include <math.h>	double modf(double, double *)	modfl() is not supported.
#include <math.h>	double pow(double x, double y), float powf(float x, float y);	powl() is not supported. Errno & return exceptions: When x=0 and y=0, EDOM is not set, and the return value is 1.0 When x=0 and y<0, EDOM is not set, and the return value is Positive Infinity When x<0 and y is non-integer, EDOM is not set, and the functions returns 0.
#include <math.h>	double sin(double), float sinf(float);	sinl() is not supported. Function does not set errno. For +-NAN input returns +-NAN respectively. For +-Infinity input returns -NAN.

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <math.h>	double sinh(double), float sinhf(float);	sinhl() is not supported. Function does not set errno. For +-NAN input returns +-NAN respectively. For +-Infinity input returns +-Infinity respectively.
#include <math.h>	double sqrt(double x), float sqrtf(float x);	sqrtl() is not supported. Errno & return exceptions: EDOM is not set for the following conditions. When x<0, the return value is -NAN. When x = +Infinity, the return value is +Infinity respectively. When x = +-NAN, the return value is +-NAN.
#include <math.h>	double tan(double x), float tanf(float x);	tanl() is not supported. Errno is not set by this function. Return exceptions: When x = +-NAN, the return value is +-NAN respectively. When x = +-Infinity, the return value is -NAN.
#include <math.h>	double tanh(double x), float tanhf(float x);	tanh() is not supported. Errno is not set by this function. Return exceptions: When x = +-NAN, the return value is +-NAN respectively.
#include <stdlib.h>	<u>Math:</u> void div_r( int numerator, int denominator, div_t * divStructPtr )  typedef struct { int quot; int rem; } div_t	div() is not supported because it is not re-entrant. Description: This routine computes the quotient and remainder of <i>numer/denom</i> . The quotient and remainder are stored in the div_t structure pointed to by <i>divStructPtr</i> . This function does not set errno. Denominator = 0 will cause a divide by 0 fault and put the CPU into CPU Halted mode.
#include <stdlib.h>		ldiv() is not supported because it is not re-entrant.
NA		_lrotr, _lrotr are not supported (long rotate left and right respectively).

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <ctkGefCLib.h>	max(a,b), min(a,b)	max(), min() macros are supported in the GefCLib library via macros in the header file. max() returns the greater of two numbers and min() returns the smaller of two numbers. These macros do not set errno.
#include <stdlib.h>	int rand(void)	This function does not set errno.
NA		_rotl, _rotr are not supported (int rotate left and right respectively).
#include <stdlib.h>	void srand(unsigned int seed)	This function does not set errno.
#include <stdlib.h>	<u>Data Conversion:</u> int abs(int)	This function does not set errno. Return exceptions: For an input value of -2147483648, the return value is -2147483648.
#include <stdlib.h>	double atof(const char *)	Sets errno if the input cannot be represented as a 64-bit floating point number. (For ex. numbers significantly outside +-1.79e308 range.  <i>Note: numbers just beyond this range will return +-Infinity but will not set errno)</i>
#include <stdlib.h>	int atoi(const char *)	Sets errno if the input cannot be represented as a 32-bit signed integer. (For example, numbers outside -2147483648 to +2147483647 range)
#include <stdlib.h>	long atol(const char *)	Sets errno if the input cannot be represented as a 32-bit signed integer. (For example, numbers outside -2147483648 to +2147483647 range)
NA		_itoa() (Convert an integer to a string) is not supported.
#include <stdlib.h>	long labs(long)	This function does not set errno.
NA		_ltoa() (Convert a long integer to a string) is not supported.
#include <stdlib.h>	long strtol(const char *, char ** endptr, int base)	Sets errno if the input cannot be represented as a 32-bit signed integer. (For example, numbers outside -2147483648 to +2147483647 range).
#include <stdlib.h>	unsigned long strtoul(const char *, char ** endptr, int base)	Sets errno if the input cannot be represented as a 32-bit unsigned integer. (For example, numbers outside -0 to 4294967295 range)

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
NA		_ultoa() (Convert an unsigned long integer to a string) is not supported
#include <stdlib.h>	<u>Search:</u> void *bsearch(const void *key, const void * base, size_t nmemb, size_t size, int (* compar)(const void *, const void *))	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <stdlib.h>	qsort(void * base, size_t nmemb, size_t size, int(* _compar)(const void *, const void *))	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
NA	<u>Search:</u>	_lfind() (Performs a linear search for the specified key). Not supported.
NA		_lsearch() (Performs a linear search for a value; adds to end of list if not found). Not supported.
#include <string.h>	<u>String Manipulation:</u> char *strcat(char *, const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	char *strchr(const char *, int)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	int strcmp(const char *, const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	char *strcpy(char *, const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <string.h>	size_t strcspn(const char *, const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	char *strerror_r(int errorcode, char *stringBuffer)	strerror() and _strerror() are not supported since they are not re-entrant Description: This routine maps the error number in errcode to an error message string. It stores the error string in <i>buffer</i> . The function returns GEF_OK or GEF_ERROR. GEF_ERROR is returned if a NULL pointer is passed as the input for stringBuffer. Errno is not set.
NA		_stricmp() (Perform a lowercase comparison of strings) is not supported.
#include <string.h>	size_t strlen(const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
NA		_strlwr() (Convert a string to lowercase) is not supported.
#include <string.h>	char *strncat(char *, const char *, size_t)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	int strncmp(const char *, const char *, size_t)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	char *strncpy(char *, const char *, size_t)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
NA		_strnicmp() (Compare characters of two strings without regard to case) is not supported.
NA		_strnset() (Initialize characters of a string to a given format. ) is not supported.



Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <string.h>	char *strpbrk(const char *, const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	char *strchr(const char *, int)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
NA		_strrev() (Reverse characters of a string) is not supported.
NA		_strset() (Set characters of a string to a character) is not supported.
#include <string.h>	size_t strspn(const char *, const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	char *strstr(const char *, const char *)	This function does not set errno.  <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <string.h>	<pre>char *strtok_r(     char * string,     const char * separators,     char ** ppLast )</pre>	<p>strtok() and _fstok() are not supported since they are not re-entrant</p> <p>Description:</p> <p>This routine considers the null-terminated string as a sequence of zero or more text tokens separated by spans of one or more characters from the separator string <i>separators</i>. The argument <i>ppLast</i> points to a user-provided pointer which in turn points to the position within <i>string</i> at which scanning should begin.</p> <p>In the first call to this routine, <i>string</i> points to a null-terminated string; <i>separators</i> points to a null-terminated string of separator characters; and <i>ppLast</i> points to a NULL pointer. The function returns a pointer to the first character of the first token, writes a null character into <i>string</i> immediately following the returned token, and updates the pointer to which <i>ppLast</i> points so that it points to the first character following the null written into <i>string</i>. (Note that because the separator character is overwritten by a null character, the input string is modified as a result of this call.)</p> <p>In subsequent calls <i>string</i> must be a NULL pointer and <i>ppLast</i> must be unchanged so that subsequent calls will move through the string, returning successive tokens until no tokens remain. The separator string <i>separators</i> may be different from call to call. When no token remains in <i>string</i>, a NULL pointer is returned. This function returns a pointer to the first character of a token, or a NULL pointer if there is no token.</p>
NA		<p>_strupr() (Converts any lowercase characters in the specified string to uppercase) is not supported.</p>

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <PLCC9070.h>	_fstrcat() _fstrchr() _fstrcmp() _fstrncpy() _fstrcspn() _fstrlen _fstrncat() _fstrncmp() _fstrncpy() _fstrpbrk() _fstrrchr() _fstrspn() _fstrstr()	_fstrcat() _fstrchr() _fstrcmp() _fstrncpy() _fstrcspn() _fstrlen _fstrncat() _fstrncmp() _fstrncpy() _fstrpbrk() _fstrrchr() _fstrspn() _fstrstr() These functions are far pointer versions of functions without the “_f” prefix. Since far pointer versions are not needed for a 32-bit architecture, PLCC9070.h equates these functions to the primary functions with the following type of statement: <b>#define _fstrcat strcat</b>
#include <PLCC9030.h>	_fstrcat() _fstrchr() _fstrcmp() _fstrncpy() _fstrcspn() _fstrlen() _fstrncat() _fstrncmp() _fstrncpy() _fstrpbrk() _fstrrchr() _fstrspn() _fstrstr() _fmemchr() _fmemcmp() _fmemcpy() _fmemmove() _fmemset()	These functions are far pointer versions of functions without the “_f” prefix. Since far pointer versions are not needed for a 32-bit architecture, PLCC9030.h equates these functions to the primary functions with the following type of statement: <b>#define _fstrcat strcat</b>
NA		_fstricmp() _fstrlwr() _fstrnicmp() _fstrnset() _fstrrev() _fstrset() _fstrtok _fstrupr <b>These functions are not supported.</b>
#include <string.h>	<u>Buffer Manipulation:</u>	
NA		_memccpy() (Copies characters from a buffer) is not supported.
#include <string.h>	void *memchr(const void *, int, size_t)	This function does not set errno. <hr/> <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <string.h>	int memcmp(const void *, const void *, size_t)	This function does not set errno. <hr/> <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <string.h>	void * memcpy(void *, const void *, size_t)	<p>This function does not set errno.</p> <hr/> <p><b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.</p>
NA		<p>_memicmp() - compares characters in two buffers (case-insensitive) - is not supported.</p>
#include <string.h>	void * memmove(void *, const void *, size_t)	<p>This function does not set errno.</p> <hr/> <p><b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.</p>
#include <string.h>	void * memset(void *, int, size_t)	<p>This function does not set errno.</p> <hr/> <p><b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.</p>
#include <GefCLib.h>	void _swab(char *source, char *destination, int nbytes)	<p>_swab() swap “nbytes” bytes from the “source” buffer (swaps even and odd bytes) and copies the result to the “destination” buffer where buffers do not have to be aligned on even byte boundaries. If “nbytes” is not an odd number, the function will swap nbytes+1. Supported in GefCLib.h with the following statement:</p> <p><b>#define _swab uswab</b></p> <p>This function does not return errno.</p> <hr/> <p><b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.</p>
#include <PLCC9070.h>	_fmemchr, _fmemcmp, _fmemcpy, _fmemmove, _fmemset	<p>_fmemchr, _fmemcmp, _fmemcpy, _fmemmove, _fmemset</p> <p>These functions are <b>far</b> pointer versions of functions without the “_f” prefix. Since far pointer versions are not needed for a 32-bit architecture, PLCC9070.h equates these functions to the primary functions with the following type of statement:</p> <p><b>#define _fmemcpy memcpy</b></p>

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <PLCC9030.h>	_fmemchr, _fmemcmp, _fmemcpy, _fmemmove, _fmemset	_fmemchr, _fmemcmp, _fmemcpy, _fmemmove, _fmemset These functions are <i>far</i> pointer versions of functions without the “_f” prefix. Since far pointer versions are not needed for a 32-bit architecture, PLCC9030.h equates these functions to the primary functions with the following type of statement: <b>#define _fmemcpy memcpy</b>
NA		_fmemccpy, _fmemicmp These functions are not supported.
#include <string.h>	<u>Internationalization:</u> int strcoll(const char *, const char *)	This function does not return errno. <hr/> <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <time.h>	<u>Time Internationalization:</u> size_t strftime(char *_s, size_t _maxsize, const char *_fmt, const struct tm *_t)	This function does not return errno. <hr/> <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <time.h>	<u>Time:</u> int asctime_r(const struct tm * timeptr, char * asctimeBuf, size_t * buflen)	asctime() is not supported since it is not re-entrant Description: This routine converts the broken-down time pointed to by <i>timeptr</i> into a string of the form: <b>SUN SEP 16 01:03:52 1973\n\0</b> The string is copied to <i>asctimeBuf</i> . This function returns the size of the created string. This function does not return errno. <hr/> <b>Note:</b> NULL or invalid input pointers to this function will put the CPU into CPU Halted mode.
#include <time.h>	double difftime(time_t _time2, time_t _time1)	This function does not set errno.
NA		_strdate() (Copy a date to a buffer) is not supported.
NA		_strtime() (Copy the time to a buffer) is not supported.

Include File	Supported C Run-Time Library Functions Associated with File	Series 90 C Run-Time Library Compatibility Notes, Issues, Errno information and return value exceptions
#include <ctype.h>	<u>Character Classification and Conversion:</u> isalnum()	
#include <ctype.h>	int isalpha(int c)	
NA		isascii() is not supported.
#include <ctype.h>	int iscntrl(int c)	
#include <ctype.h>	int isdigit(int c)	
#include <ctype.h>	int isgraph(int c)	
#include <ctype.h>	int islower(int c)	
#include <ctype.h>	int isprint(int c)	
#include <ctype.h>	int ispunct(int c)	
#include <ctype.h>	int isspace(int c)	
#include <ctype.h>	int isupper(int c)	
#include <ctype.h>	int isxdigit(int c)	
#include <ctkGefCLib.h>	int toascii(int c)	
#include <ctype.h>	int tolower(int c)	_tolower() is not supported; use tolower()
#include <ctype.h>	int toupper(int c)	_toupper() is not supported use toupper().

## Appendix C: Diagnostics

This section includes descriptions of some known problems and solutions to those problems.

### C-1 **Issue: Compiler issues the following warning when the EnableAnsi flag is used:**

myCBlock.c:240: warning: implicit declaration of function `infinity'  
myCBlock.c:263: warning: implicit declaration of function `acosf'  
myCBlock.c:264: warning: implicit declaration of function `asinf'  
myCBlock.c:265: warning: implicit declaration of function `atanf'  
myCBlock.c:266: warning: implicit declaration of function `ceilf'  
myCBlock.c:267: warning: implicit declaration of function `cosf'  
myCBlock.c:268: warning: implicit declaration of function `coshf'  
myCBlock.c:269: warning: implicit declaration of function `expf'  
myCBlock.c:270: warning: implicit declaration of function `fabsf'  
myCBlock.c:271: warning: implicit declaration of function `floorf'  
myCBlock.c:272: warning: implicit declaration of function `fmodf'  
myCBlock.c:273: warning: implicit declaration of function `logf'  
myCBlock.c:274: warning: implicit declaration of function `log10f'  
myCBlock.c:275: warning: implicit declaration of function `powf'  
myCBlock.c:276: warning: implicit declaration of function `sinf'  
myCBlock.c:277: warning: implicit declaration of function `sinhf'  
myCBlock.c:278: warning: implicit declaration of function `sqrtf'  
myCBlock.c:279: warning: implicit declaration of function `tanf'  
myCBlock.c:280: warning: implicit declaration of function `tanhf'

Solution: The warnings are given because these are not supported ANSI functions. However, if you choose, you can store the C Block to the PLC because these functions are supported in the PLC. To get rid of the warnings, compile the C Block without the EnableAnsi flag.

## **C-2      Issue: Compiler issues the following statement: warning: `HUGE\_VAL' redefined.**

Solution: Place the PACRXPLC.h, PACRX3iPlc.h, PACRX7iPlc.h or PACRSTiPlc.h include file before math.h. This properly defines HUGE\_VAL and prevents redefinition. If the warning is ignored, the C Block may not store successfully to the PLC due to not being able to resolve a reference used by HUGE\_VAL.

## **C-3      Issue: Compiler issues the following error statement: undefined reference to `isascii' when the EnableAnsi flag is used. In addition, the C Block will not store to the PLC.**

Solution: The isascii macro is not supported when compiling with ANSI checking turned on. If the function is required, you will need to compile without the EnableAnsi flag. The C Block will not store because there is not a isascii function in the PLC to link with the symbol.

## **C-4      Issue: On some Windows 2000 PCs, the local DOS Box Environment "path" variable is not used, resulting in the compile process failing because the path to the compiler batch file is not found.**

Solution: The problem can be corrected using the following steps:

1. Press Start->Settings->Control Panel
2. Double click on System
3. Click on the "Advanced Tab"
4. Click on the "Environment Variables" button
5. In the System Variables window, scroll to the "Path" variable and click on it to highlight it.
6. Press the Edit button.
7. Add the following text at the end of the current string



```
”;<PACSystemsInstallLocation>\Compilers\ElfX86;  
<PACSystemsInstallLocation>\Compilers\CommonTools;  
<PACSystemsInstallLocation>\Targets\PACRX\Compiler;  
<PACSystemsInstallLocation>\Targets\DebugPACRX\Compiler;  
<PACSystemsInstallLocation>\Targets\PACRX3i\Compiler;  
<PACSystemsInstallLocation>\Targets\DebugPACRX3i\Compiler;  
<PACSystemsInstallLocation>\Targets\PACRX7i\Compiler;  
<PACSystemsInstallLocation>\Targets\DebugPACRX7i\Compiler;  
<PACSystemsInstallLocation>\Targets\PACRSTi\Compiler;  
<PACSystemsInstallLocation>\Targets\DebugPACRSTi\Compiler;  
<PACSystemsInstallLocation>\Targets\CommonFiles\CompilerCommon"
```

where <PACSystemsInstallLocation> is the location of the C Toolkit installation on your machine. For example, the default installation location is: C:\GE Software\PACSystemsCToolkit

8. Press OK three times to exit from the System Properties application
9. Reboot your PC.

## Technical Support & Contact Information:

Home link: <http://www.Emerson.com/Industrial-Automation-Controls>

Knowledge Base: <https://www.emerson.com/Industrial-Automation-Controls/support>

**Note:** If the product is purchased through an Authorized Channel Partner, please contact the seller directly for any support.

Emerson reserves the right to modify or improve the designs or specifications of the products mentioned in this manual at any time without notice. Emerson does not assume responsibility for the selection, use or maintenance of any product. Responsibility for proper selection, use and maintenance of any Emerson product remains solely with the purchaser.

© 2019 Emerson. All rights reserved.

Emerson Terms and Conditions of Sale are available upon request. The Emerson logo is a trademark and service mark of Emerson Electric Co. All other marks are the property of their respective owners.

