Telepace C Tools

User and Reference Manual

5/12/2011

Schneider Electric

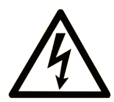
Table of Contents

Telepace C Tools Structures and Types 506

Index of Figures

Safety Information

Read these instructions carefully, and look at the equipment to become familiar with the device before trying to install, operate, or maintain it. The following special messages may appear throughout this documentation or on the equipment to warn of potential hazards or to call attention to information that clarifies or simplifies a procedure.

	The addition of this symbol to a Danger or Warning safety label indicates that an electrical hazard exists, which will result in personal injury if the instructions are not followed.

	This is the safety alert symbol. It is used to alert you to potential personal injury hazards. Obey all safety messages that follow this symbol to avoid possible injury or death.

⚠DANGER
DANGER indicates an imminently hazardous situation which, if not avoided, **will result in** death or serious injury.

⚠WARNING
WARNING indicates a potentially hazardous situation which, if not avoided, **can result** in death or serious injury.

⚠CAUTION
CAUTION indicates a potentially hazardous situation which, if not avoided, **can result** in minor or moderate.

CAUTION
CAUTION used without the safety alert symbol, indicates a potentially hazardous situation which, if not avoided, **can result** in equipment damage..

PLEASE NOTE

Electrical equipment should be installed, operated, serviced, and maintained only by qualified personnel. No responsibility is assumed by Schneider Electric for any consequences arising out of the use of this material.

A qualified person is one who has skills and knowledge related to the construction and operation of electrical equipment and the installation, and has received safety training to recognize and avoid the hazards involved.

BEFORE YOU BEGIN

Do not use this product on machinery lacking effective point-of-operation guarding. Lack of effective point-of-operation guarding on a machine can result in serious injury to the operator of that machine.

⚠CAUTION
UNINTENDED EQUIPMENT OPERATION • Verify that all installation and set up procedures have been completed. • Before operational tests are performed, remove all blocks or other temporary holding means used for shipment from all component devices. • Remove tools, meters, and debris from equipment **Failure to follow these instructions can result in death, serious injury or equipment damage.**

Follow all start-up tests recommended in the equipment documentation. Store all equipment documentation for future references.

Software testing must be done in both simulated and real environments.

Verify that the completed system is free from all short circuits and grounds, except those grounds installed according to local regulations (according to the National Electrical Code in the U.S.A, for instance). If high-potential voltage testing is necessary, follow recommendations in equipment documentation to prevent accidental equipment damage.

Before energizing equipment:

- Remove tools, meters, and debris from equipment.
- Close the equipment enclosure door.
- Remove ground from incoming power lines.
- Perform all start-up tests recommended by the manufacturer.

OPERATION AND ADJUSTMENTS

The following precautions are from the NEMA Standards Publication ICS 7.1-1995 (English version prevails):

- Regardless of the care exercised in the design and manufacture of equipment or in the selection and ratings of components, there are hazards that can be encountered if such equipment is improperly operated.
- It is sometimes possible to misadjust the equipment and thus produce unsatisfactory or unsafe operation. Always use the manufacturer's instructions as a guide for functional adjustments. Personnel who have access to these adjustments should be familiar with the equipment manufacturer's instructions and the machinery used with the electrical equipment.
- Only those operational adjustments actually required by the operator should be accessible to the operator. Access to other controls should be restricted to prevent unauthorized changes in operating characteristics.

About The Book

At a Glance

Document Scope

This manual describes the Telepace C Tools programming for the SCADAPack 16-bit controllers.

Validity Notes

This document is valid for all versions of firmware for the SCADAPack 16-bit controllers.

Product Related Information

⚠WARNING
UNINTENDED EQUIPMENT OPERATION The application of this product requires expertise in the design and programming of control systems. Only persons with such expertise should be allowed to program, install, alter and apply this product. Follow all local and national safety codes and standards. **Failure to follow these instructions can result in death, serious injury or equipment damage.**

User Comments

We welcome your comments about this document. You can reach us by e-mail at technicalsupport@controlmicrosystems.com.

Telepace C Tools Overview

The Telepace C Tools are ideal for engineers and programmers who require advanced programming tools for SCADA applications and process control. The SCADAPack, 4000 controllers execute ladder logic and C application programs simultaneously, providing you with maximum flexibility in implementing your control strategy.

This manual provides documentation on the Telepace C program loader and the library of C language process control and SCADA functions. We strongly encourage you to read it.

We sincerely hope that the reliability and flexibility afforded by this fully programmable controller enable you and your company to solve your automation applications in a cost effective and efficient manner.

The Telepace C Tools include an ANSI C cross compiler; a customized library of functions for industrial automation and data acquisition; a real time operating system; and the Telepace C program loader. The C function library is similar to many other C implementations, but contains additional features for real time control, digital and analog I/O. An overview of the application development environment and its features follows.

Program Development

C programs are written using any text editor. The MCCM77 compiler is used to compile, assemble and link the programs on a personal computer.

The memory image, which results from this process may then be, loaded either into the RAM, committed to an EPROM, or both may be used together. Programs may be executed either manually or automatically at power up.

Modularity

Programs written in Telepace C may be split into many separately compiled modules. These modules may be tested individually before being linked together in the final program. Command files specify how the various files are to be linked.

Assembly Language Code

Assembly language source code may be included directly within C programs. The #asm and #endasm statements are used to enclose in-line assembly language code, which is then assembled without passing through the compiler.

C programs are converted to assembly language by the MCCM77 compiler, and this code may be viewed and modified. The resulting code may also be combined with programs written directly in assembler.

Program Options

A C application program may reside in RAM or ROM. The normal method of program development has the program in RAM. The program may call library routines in the operating system ROM. The RAM is nonvolatile (battery backed), so the program may remain in RAM once development is completed and the unit is installed.

Application programs may also be committed to EPROM. The RAM is used for data storage in this case.

Supported Language Features

The Telepace C Tools use the Microtec® MCCM77 C compiler. The compiler is ANSI C compliant, and provides a code optimizer and assembler.

In addition to the standard C operators, data types and library functions, the C tools provide a set of routines specifically designed for control applications. Some applications and the descriptions of these functions may be found on the following pages.

Serial Communication

An extensive serial communication library supports simple ASCII communication, communication protocols and serial port configuration. The default communication mode uses the TeleBUS RTU communication protocol. It supports access to the I/O database, serial port reconfiguration and program loading.

The application program can disable the TeleBUS protocol, and use the serial ports for other purposes.

TeleBUS protocols are compatible with the widely supported, Modbus ASCII and RTU protocols.

Clock/Calendar

The processor's hardware clock calendar is supported by the C Tools. The time, date and day of week can be read and set by the application software.

Timers

The controller provides 32 software timers. They are individually programmable for tick rates from ten per second to once every 25.5 seconds. Timers may be linked to digital outputs to cause external devices to turn on/off after a specified period. Timers operate in the background from a hardware interrupt generated by the main system clock.

Duty Cycle and Pulse Outputs

The digital I/O driver provides duty cycle and pulse train outputs. Duty cycle outputs generate continuous square waves. Pulse train outputs generate finite sequences of pulses. Outputs are generated independent of the application program.

Watchdog Timer

The controller supports a hardware watchdog timer to detect and respond to hardware or software failures. Watchdog timer trigger pulses may be generated by the user program or by the system clock.

Checksums

To simplify the implementation of self-checking communication algorithms, the C Tools provide four types of checksums: additive, CRC-16, CRC-CCITT, and byte-wise exclusive-OR. The CRC algorithms are particularly robust, employing various polynomial methods to detect communication errors. Additional types of checksums are easily implemented using library functions.

Standard I/O Functions

The Telepace C Tools are an enhanced version of standard C libraries. Many of the usual C programming techniques apply. However, with respect to I/O, there are some differences.

The C Tools function library supports the standard I/O functions. There are no disk-drives or peripherals associated with the controller. Thus many file handling functions return fixed responses, indicating that the operation could not be performed.

Standard devices are opened automatically by the operating system and cannot be closed. The route function may be used to redirect stdin, stdout and stderr.

The Telepace Program

Telepace is an easy-to-use interface providing, among several other features, a C Program Loader and a Ladder Logic program editor. On-line help provides a reference to the features of the Telepace program. Telepace runs on the Microsoft Windows operating system.

This manual references only those features of Telepace pertaining to the C Program Loader dialog. Please refer to the section ***Telepace Program Reference*** for a complete description of Telepace menus, which will be useful during C Program development.

Additional Documentation

Additional documentation on Telepace Ladder Logic and the SCADAPack controllers is found in the following documents.

The on-line help for the Telepace C program loader contains a complete reference to the operation of the loader. To display on-line help, select **Contents** from the **Help** menu.

The ***SCADAPack & Micro16 System Manual*** is a complete reference to controller and I/O modules used with SCADAPack and Micro16 controllers. It contains the ***SCADAPack Controller Hardware Manual,*** the ***Micro16 System Manual*** and hardware manuals for 5000 I/O modules.

The ***Telepace Ladder Logic Reference and User Manual*** describes the creation of application programs in the Ladder Logic language.

The ***TeleBUS Protocols User Manual*** describes communication using Modbus compatible protocols.

The ***Telepace PID Controller Reference Manual*** describes PID control concepts and provides examples using the PID functions.

Getting Started

This section of the C Tools User Manual describes the installation of C Tools and includes a Program Development Tutorial. The Program Development Tutorial leads the user through the steps involved in writing, compiling, linking and loading a C application program.

System Requirements

Telepace requires the following minimum system configuration.

- Personal computer using 80386 or higher microprocessor.
- Microsoft Windows™ operating system versions including Windows 2000, NT and XP™ .
- Minimum 4 MB of memory.
- Mouse or compatible pointing device.
- Hard disk with approximately 2.5 Mbytes of free disk space.

Making Backup Disks

You should make a backup copy of the Telepace disk and Microtec C compiler disks before using the software. Work with the backup copy – if it becomes unusable you can make a new copy from the original disk.

- In My Computer, click the icon for the disk you want to copy.
- On the File menu, click Copy Disk.
- Click the drive you want to copy from and the drive you want to copy to, and then click Start.

Installation of C Compiler

Install the Microtec C compiler as described in the installation manuals supplied with the system.

To run the Microtec Compiler and Linker from any directory, without the need to specify the full path, you will have to setup the following System Environmental Variables:

Variable	**Value**
mri_m77_bin	c:\mccm77;c:\asmm77
mri_m77_inc	c:\mccm77
mri_m77_lib	c:\mccm77
mri_m77_tmp	c:\mccm77\tmp

In addition you would need to add these values to the Path System Variable:

C:\MCCM77;C:\ASMM77;C:\XHSM77

Spaces are not tolerated in between entries in the Path value.

On a Windows XP Control Panel, select **System | Advanced | Environmental Variables** to access the dialog where the above variables need to be set.

Installation of Telepace

Install Telepace as described in the installation section of the *Telepace Ladder Logic Reference and User Manual.*

Some virus checking software may interfere with Setup. If you experience difficulties with the Setup, disable your virus checker and run Setup again.

Installing C Tools as an Upgrade

If you are installing Telepace as an upgrade to a previous C Tools installation for the Micro16, the C Tools are installed in the new directory c:\Telepace\ctools\520x instead of the directory c:\Telepace\ctools\micro16.

If the older version of C Tools is not needed, copy user data files out of the micro16 directory and delete the directory and its contents.

When linking older programs you will need to modify older linker command (.cmd) files to reference the new 520x directory instead of the micro16 directory, or see the sample linker file appram.cmd for the correct file contents.

The sample linker command file appram.cmd also loads the new ctools.lib library. This library contains the new C Tools functions defined in the header file ctools.h.

Program Development Tutorial

Program development consists of three stages: writing and editing; compiling and linking; and loading the program into the controller. Each uses separate tools. To demonstrate these steps a sample program will be prepared.

Refer to the **C Program Development** section for a description of the program development process.

Traditionally, the first program that is run on a new C compiler is the *hello, world* program. It prints the message “hello, world”.

Writing and Editing

A controller C program is written using any text editor or word processor in text mode. The syntax should correspond to that described in the *Microtec MCCM77 Documentation Set*, and the ***C Program Development*** section of this manual. This chapter describes non-standard functions, which are unique to the controller. It should be read carefully to make use of the special purpose routines available.

Using your text editor, open the file hello.c file. It is located in the Telepace\ctools\520x directory. The program looks a little different from the traditional *hello, world* program.

```
/* --------------------------------------------------
   hello.c
   SCADAPack and Micro16 Test Program

   The infamous hello, world program.
   ----------------------------------------------- */

#include <ctools.h>

void main(void)
{
       PROTOCOL_SETTINGS settings;

       /* Disable the protocol on serial port 1 */
       settings.type        = NO_PROTOCOL;
       settings.station     = 1;
       settings.mode        = AM_standard;
       settings.priority    = 3;
       settings.SFMessaging = FALSE;
       setProtocolSettings(com1, &settings);

       /* Print the message */
       fprintf(com1, "hello, world\r\n");

       /* Wait here forever */
       while (TRUE)
       {
              NULL;
       }
}
```

The “hello, world” message will be output to the *com1* serial port of the controller. A terminal connected to the port will display the message.

The controller normally communicates on all ports using the TeleBUS communication protocol. The first section of the program disables the *com1* protocol so the serial port can be used as a normal RS-232 port.

The fprintf function prints the message to the com1 serial port.

When you have completed examining the program, close the hello.c file. It is now ready to be compiled and linked.

Compiling and Linking

Compiling and linking convert the source code into executable code for the controller. The Telepace C Tools use a C cross compiler and linker from Microtec, a respected supplier of embedded system tools. The compiler produces tight, well-optimized code. The compiler and linker run under the Microsoft MS-DOS operating system.

The compiler has many command line options. The basic command line and options required to compile code for the controller are:

mccm77 -v -nQ -Ml -c *filename*.c

This should be repeated for each file in the application. The command line options are case sensitive. The character following the M is a lower case l (ell).

Files are linked together using linker command files. To link a program execute the command:

lnkm77 -c *filename*.cmd

Sample command files for RAM and ROM based applications are located in the Telepace\ctools\520x directory.

Example

The hello.c program is found in the Telepace\ctools\520x directory. To compile and link the program:

- switch to the Telepace\ctools\520x directory;
- enter the commands

```
mccm77 -v -nQ -Ml -c hello.c
lnkm77 -c hello.cmd
```

The file hello.abs contains the executable code in a format ready to load into the controller.

Loading and Executing

The Telepace C Program Loader transfers executable files from a PC to the controller and controls execution of programs in the controller. The loader can also initialize program memory and serial port configuration.

Controller Initialization

The memory of the controller has to be initialized when beginning a new programming project or when it is desired to start from default conditions. It is not necessary to initialize the controller before every program load.

To initialize the controller, first perform a SERVICE boot. A SERVICE boot preserves programs and data in nonvolatile RAM, but does not start the programs running. Default communication parameters are used.

To perform a service boot:

- Remove power from the controller.
- Press and hold the LED POWER switch.
- Apply power to the controller.
- Wait until the STAT LED on the top of the board turns on.
- Release the LED POWER switch.

Second, initialize the program and data memory in the controller. A new controller will require initializations to be performed. Selected initializations can be performed on a controller that is in use.

- Run the Telepace program under Microsoft Windows.
- Connect the PC to the controller with the appropriate serial cable. The *hello, world* program will print data on the *com1* serial port. Therefore connect to the *com2* serial port on the controller. (All communication ports work the same. We use *com2* here because the sample program is using *com1.)*
- From the **Controller** menu, select under **Type** the controller type that is connected. A check mark appears beside the desired type when it is selected.
- From the **Controller** menu, select the **Initialize** command.
- Select all options: **Erase Ladder Logic Program, Erase C Program**, **Initialize Controller** and **Erase Register Assignment Table**.
- Click on the **OK** button.

The controller is now ready for a program.

Loading the Program

To load the *hello, world* program into the controller:

- Run the Telepace program.
- From the **Controller** menu, select the **C Program Loader** command.
- Enter hello.abs in the edit box for the C Program file name.
- Select all write options: **C Program**, **Register Assignment** and **Serial port settings**.
- Click on the **Write** button. The file will be downloaded.
- A message about the empty register assignment will appear. Click on the **OK** button.

Executing the Program

- Connect a terminal to *com1* on the controller. It will display the output of the program. Set the communication parameters to 9600 baud, 8 data bits, 1 stop bit, and no parity.
- From the **C Program Loader** dialog, click on the **Run** button to execute the program. The “hello, world” message will be displayed on the terminal.

Serial Communication Parameters

When the controller is powered up in the SERVICE mode the serial ports are configured as:

- 9600 baud
- 8 data bits

- 1 stop bit
- no parity
- Modbus RTU protocol emulation
- station address = 1

A program may change these settings with the set_port function. When the controller is powered up in RUN position, the custom parameters, as stored by the most recent save function, are used.

C Program Development

Program Architecture

A C application program may be contained in a single file or in a number of separate files, called modules. A single file is simple to compile and link. It can become cumbersome to edit and time-consuming to compile as the file grows in size.

An application stored in separate modules by function is easier to edit, promotes function re-use, and is quicker to compile when only a few modules are changed. Compiled modules can be combined into object libraries and shared among users.

The Telepace C Tools support both single file and multiple module programs. A C application program consists of support functions provided by the C Tools and the main() and other functions written by the user.

Main Function Structure

The program sample below shows a typical structure for the main() function.

```
void main(void)
{
	/* Perform initialization actions */
	/* Start support tasks */

	/* Main Loop*/
	while (TRUE)
	{
		/* Perform application functions */
	}
}
```

Initialization actions typically consist of variable declarations, variable initialization and one-time actions that need to be performed when the program starts running.

Supporting tasks (see ***Real Time Operating System*** section) are typically created before the main loop of the program. Tasks can be created and ended dynamically during the execution of a program as well.

The main loop of a program is always an infinite loop that continually performs the actions required by the program. The main() function normally never returns.

Example

The following is an example of a three-module program. Each function is stored in a separate file. This program will be used in subsequent examples.

File: func1.c

```
#include <ctools.h>

void func1(void)
{
	fputs("This is function 1\r\n", com1);
}

File: func2.c
#include <ctools.h>

void func2(void)
{
	fputs("This is function 2\r\n ", com1);
}

File: main.c
#include <ctools.h>

extern void func1(void);
extern void func2(void);

void main(void)
{
	func1();

	while (TRUE)
	{
		func2();
	}
}
```

Start-Up Function Structure

The user’s main() function is called from the appstart function of the C Tools. It is not necessary to understand the appstart function to write programs. However it performs a number of useful functions that can be modified by the user.

The start-up code has five major functions:

- create and initialize the application program heap (for dynamic memory allocation);
- specify the number of stack blocks allocated to the main task;
- initialize application program variables;
- control execution of the protocol, ladder logic and background I/O tasks;
- execute the main function.

Source code for the function is supplied with the C Tools. The following discussion refers to statements found in the file appstart.c.

The heap is a section of memory used by dynamic memory allocation functions such as malloc. The heap starts at the end of RAM used by the program and continues to the end of physical RAM. The limit is set by the statement:

```
end_of_heap     .EQU    41ffffh
```

The limit is set by default to the smallest memory option available for the controller. If your controller has more memory, change the value of the constant according to the following table.

RAM Installed	C Application Program RAM Addresses
128 Kbytes	none (ladder logic only)
256 Kbytes	400000h – 41FFFFh
640 Kbytes	400000h – 47FFFFh
1024 Kbytes	388000h – 3E7FFFh 400000h – 47FFFFh

The application program signature section of the file contains a constant that determines the size of the stack allocated to the main task. The stack size is sufficient for most applications. It can be changed by modifying the statement:

```
.WORD 4         ;stack size in blocks
```

Refer to the ***Real Time Operating System*** section for more information on the stack required by tasks.

The appstart function begins by initializing the heap pointers, setting all non-initialized variables to zero, and initializing system variables.

It then starts the communication protocols for each serial port, according to the stored values in the EEPROM (or the standard values on a SERVICE boot). If your application program never uses the communication protocols, some or all of the following commands can be removed, to free the stack space used by the protocol tasks.[1]

start_protocol(com1);

start_protocol(com2);

start_protocol(com3);[2]

start_protocol(com4);[3]

The background I/O task is required for the timer functions, dial-up modem communications, and PID controller functions to operate. If these functions are

[1] Stack space is required to create additional tasks. Refer to the create_task function for more information.
[2] com3 is used only in the SCADAPack and SCADAPack PLUS controllers.
[3] com4 is used only in the SCADAPack LIGHT and SCADAPack PLUS controllers.

not used, you can reduce the CPU load by changing TRUE to FALSE in the following statement:

```
runBackgroundIO(TRUE);
```

The ladder logic interpreter is required for ladder logic programs. If you are not using ladder logic, you can reduce the CPU load by changing TRUE to FALSE in the following statement:

```
runLadderLogic(TRUE);
```

The final operation is execution of the main function. The _initcopy function copies the initial values for initialized variables from the __INITDATA section in the program to the variables. If there are no errors in the data then the user's application program runs. (An error is likely only if the program in RAM has been damaged or improperly linked.)

```
if (_initcopy() == 0)
{
    main();
}
```

If the main function returns, the task is ended. First, any modem control sessions started by the application are terminated.

```
abortAllDialupApps();
```

Then the task is ended. This will cause all other APPLICATION tasks created by main to be stopped as well.

```
taskStatus = getTaskInfo(0);
end_task(taskStatus.taskID);
```

Data Storage

All non-initialized variables (local and global) are initialized to zero on program startup by the Microtec C Compiler. The I/O database is the only section of memory that is not initialized to zero on startup. Data stored in the I/O database is maintained when power to the controller is lost, and remains until the controller is initialized from the Telepace program.

In most cases the I/O database provides adequate space for data storage. However, if additional non-initialized memory is required, for example for an array of custom data structures, an non-initialized section of memory can be created as shown in the example below.

```
/* --------------------------------------------------------------
-
datalog.c
```

```
This file contains the global variable definitions for a
datalogger database.

These global variables are placed in a non-initialized section
called "savedata". All data in these variables will be maintained
over powerup.
------------------------------------------------------------------
*/
#include <datalog.h>

/* define a non-initialized section called savedata */
#pragma option -NZsavedata
#pragma option -Xp

/* Global variable definitions */

/* log index */
unsigned     logIndex;

/* log database */
struct dataLog      logData[DATA_LOG_SIZE];
```

Any variable defined in this file datalog.c will be placed in the non-initialized section arbitrarily named savedata. Code operating on these variables should be placed in a separate file, which references these global variables through external definitions placed in a header file (e.g. datalog.h).

The #pragma option directive is documented in the ***Microtec MCCM77 Documentation Set.***

Compiling Source Code

The C Compiler converts source code into object files. The basic command line and options required to compile code for the controller are:

mccm77 -v -nQ -Ml -c *filename*.c

A complete description of the command line options is given in the Microtec *MCCM77 User's Guide*. The options used here are:

Option	Description
-v	Issue warnings for features in source file. This option allows you to detect potential errors in your source code before running the program.
-nQ	Do not suppress diagnostic messages. This option provides additional warnings that allow you to detect potential errors in your source code before running the program.
-Ml	Compile for large memory model (note that the character following the M is a lower case ell).
-c	Compiler output is an object file.

The following options may be useful.

Option	Description
-Jdir	Specify the directory containing the standard include files. Adding -Jc:\Telepace\ctools\520x to the command line allows you to locate your application program files in a different directory. This helps in organizing your files if you have more than one application program.
-O	Enable standard optimizations. This produces smaller and faster executable code.
-Ot	Optimize in favor of execution time rather than code size where a choice can be made.
-nOc	Pop the stack after each function call. This increases code size and execution time. This option should only be used if there is a large number of *consecutive* function calls in your program. A large number of consecutive calls requires a large stack allocation for a task. Since the number of stack blocks is limited, using this option can reduce the stack requirements for a task. See the description for the create_task function for more information.

Each module in an application should be compiled to produce an object file. The object files are then linked together to form an executable program.

Example

The following commands are required to compile the program described in the previous sections.

```
mccm77 -v -nQ -Ml -c main.c
mccm77 -v -nQ -Ml -c func1.c
mccm77 -v -nQ -Ml -c func2.c
```

This produces three output files: main.obj; func1.obj and func2.obj. In the next section these object files will be combined into an executable program.

Linking Object Files

The linker converts object files and object file libraries into an executable program. The basic command line and options to link a program are:

lnkm77 -c *filename*.cmd

Controller programs can execute from RAM, Flash or ROM. The linker command file determines the location of the program.

RAM Based Applications

A sample linker command file for a RAM based program is appram.cmd located in the Telepace\ctools\520x directory.

The file begins by specifying the location and order of memory sections. The far_appcode section is the first section in all controller C programs. It contains the start-up code that calls the main() function. In a RAM based program, the start-up code is located at the start of C application program RAM. This address is fixed at 00400000h.

The order commands specify the order of the sections. The sections are grouped so all the code and static data sections are first. The variable data sections follow. The heap is the last section. It is allowed to grow from the end of the program data to the end of memory (see ***Start Up Function Structure*** section for more information).

The sections may be rearranged, and new sections added, according to the following rules:

- The far_appcode section needs to be first in the order listing.
- Code sections must follow the far_appcode section.
- The far_endcode section needs to be the last code section.
- Data sections must follow the code sections.
- The heap section needs to be last in the order listing.

```
; -----------------------------------------------
; Specify location and order of memory sections
; -----------------------------------------------
sect far_appcode = 00400000h
order far_appcode, far_code, (CODE), const
order strings, literals, __INITDATA, far_endcode
order far_zerovars, far_initvars, (DATA), heap
```

The next section of the command file creates initialized data sections. All variables in the specified section are initialized at start-up of the program. The linker creates a copy of the data in these sections and stores it in the __INITDATA section.

```
; -----------------------------------------------
; Create initialized variables section
; -----------------------------------------------
initdata far_initvars
```

The next section of the command file lists the application program object modules (files) to be included in the program. You may also include libraries of functions you create here. The sample command file includes one object module: app.obj.

```
; -----------------------------------------------
; Load application program object modules
; -----------------------------------------------
load app
```

The next section of the command file lists the start-up routines and standard libraries to be included. There are three object modules and two libraries:

Module	Description
Appstart.obj	This file contains the application program start up routine (see *Program Architecture* section above). If you modify

Module	Description
	the start-up routine for a particular application, specify the path to the modified routine.
Romfunc.obj	This file contains addresses of the jump table for calling functions in the operating system ROM. Only the symbols are loaded as only the addresses are needed.
Ctools.lib	This is the C Tools library, which contains C Tools functions not found in the operating system ROM.
cm77islf.lib	This is the standard Microtec floating point library.
cm77islc.lib	This is the standard Microtec function library.

```
; ----------------------------------------------
; Load start up and library routines
; ----------------------------------------------
load c:\Telepace\ctools\520x\appstart
load_symbols c:\Telepace\ctools\520x\romfunc
load c:\Telepace\ctools\520x\ctools.lib
load c:\mccm77\cm77islf.lib
load c:\mccm77\cm77islc.lib
```

The final section of the command file specifies the output file format. The listmap command specifies what information is to be included in the map file. Refer to the Microtec manuals for more information on map files.

The format command specifies the executable output will be in Motorola S2 record format. The Telepace C Program Loader requires this format.

```
; ----------------------------------------------
; Specify output file formats and options
; ----------------------------------------------
listmap nopublics, nointernals, nocrossref
format S2
```

Example

The standard command file needs to be modified to link the application described in the previous example. Copy the appram.cmd file to myapp.cmd. Modify the application object modules section to read:

```
; ----------------------------------------------
; Load application program object modules
; ----------------------------------------------
load main
load func1
load func2
```

Link the file with the command

lnkm77 -c myapp.cmd

This will produce one output file: myapp.abs. The next step is to load it into the controller using the Telepace C Program Loader.

Flash Based Applications

A sample command file for a Flash based program is appflash.cmd located in the Telepace\ctools\520x directory. This file is very similar to the command file for RAM based programs.

The file begins by specifying the location and order of memory sections. There are two types of sections in a Flash based program. The code and static data sections need to be stored in the Flash. The variable data sections need to be stored in RAM.

The far_appcode section is the first code section in all controller C programs. In a Flash based program, the far_appcode section is located at 110000h.

The far_zerovars section is the first data section. In a ROM based program it is normally located at the start of application program RAM (00400000h). It is possible to start this section at any RAM address, if your application requires it.

The order commands specify the order of the sections. The sections may be rearranged, and new sections added, according to the following rules:

- The far_appcode section needs to be first in the order listing.
- Code sections must follow the far_appcode section.
- The far_endcode section needs to be the last code section.
- The far_zerovars section needs to be the first data section.
- All other data sections need to follow the far_zerovars section.
- The heap section needs to be the last data section.

```
; ---------------------------------------------
; Specify location and order of memory sections
; ---------------------------------------------
sect far_appcode  = 00110000h
sect far_zerovars = 00400000h
order far_appcode, far_code, (CODE), const
order strings, literals, __INITDATA, far_endcode
order far_zerovars, far_initvars, (DATA), heap
```

The remaining sections of the file are identical to the RAM command file. Refer to the ***RAM Based Applications*** section for a description.

The final section of the command file specifies the output file format. The default format command specifies the executable output will be in Motorola S2 record format. This is the format required by the Telepace Flash Loader.

Example – C Program in Flash

The standard command file needs to be modified to link the application described in the previous example. Copy the appflash.cmd file to myapp.cmd. Modify the application object modules section to read:

```
; ---------------------------------------------
; Load application program object modules
; ---------------------------------------------
load main
load func1
load func2
```

Link the file with the command

```
lnkm77 -c myapp.cmd
```

This will produce one output file: myapp.abs. The next step is to write the file to the controller using Telepace. Use the Flash Loader command on the Controller menu. Consult the Telepace documentation for details.

ROM Based Applications

A ROM based program is very similar to a Flash based application. However, an EPROM programmer needs to be used to create the ROM. ROM based programs have access to more program (code) memory than Flash based applications.

It is recommended that Flash based programs be used, unless there is not enough program memory available.

If a ROM based program is created it should be stored in an EPROM. If a Flash based part is used the commands in Telepace to erase Flash may not work as expected. Contact Control Microsystems for more information about this.

A sample command file for a ROM based program is approm.cmd located in the Telepace\ctools\520x directory. This file is very similar to the command file for Flash based programs.

The file begins by specifying the location and order of memory sections. There are two types of sections in a ROM based program. The code and static data sections need to be stored in the ROM. The variable data sections must be stored in RAM.

The far_appcode section is the first code section in all controller C programs. In a ROM based program, the far_appcode section can be located at any address that is a multiple of 100h in the application ROM. The start of application ROM is fixed at 100000h.

A C application program may share the application ROM space with a ladder logic program. If only a C program is stored in the ROM the far_appcode section is located at 100000h. If a ladder logic program is stored in ROM it needs to start at 100000h. The C application can start anywhere after the end of the ladder logic program. This location is determined by the size of the ladder logic program and is determined by examining the memory image of the ladder logic program in your EPROM programmer.

The far_zerovars section is the first data section. In a ROM based program it is normally located at the start of application program RAM (00400000h). It is possible to start this section at any RAM address, if your application requires it.

The order commands specify the order of the sections. The sections may be rearranged, and new sections added, according to the following rules:

- The far_appcode section needs to be first in the order listing.
- Code sections need to follow the far_appcode section.
- The far_endcode section needs to be the last code section.
- The far_zerovars section needs to be the first data section.
- All other data sections need to follow the far_zerovars section.
- The heap section must be the last data section.

```
; ----------------------------------------------
; Specify location and order of memory sections
;
; Note: the far_appcode section address must
;       be a multiple of 100h.
; ----------------------------------------------
sect far_appcode  = 00100000h
sect far_zerovars = 00400000h
order far_appcode, far_code, (CODE), const
order strings, literals, __INITDATA, far_endcode
order far_zerovars, far_initvars, (DATA), heap
```

The remaining sections of the file are identical to the RAM command file. Refer to the ***RAM Based Applications*** section for a description.

The final section of the command file specifies the output file format. The default format command specifies the executable output will be in Motorola S2 record format. Your EPROM programmer may require a different output format. The following options are available. Refer to the Microtec Linker manual for a complete description.

Command	Description
format ASCII	Intel ASCII hex format. This is also known as Intel-86 or Extended Intel Hex format.
format IEEE	Microtec extended IEEE-695 format.
format S1	Motorola S1 record format.
format S2	Motorola S2 record format.

Example – C Program in ROM

The standard command file needs to be modified to link the application described in the previous example. Copy the approm.cmd file to myapp.cmd. Modify the application object modules section to read:

```
; ----------------------------------------------
; Load application program object modules
; ----------------------------------------------
load main
load func1
load func2
```

Link the file with the command

lnkm77 -c myapp.cmd

This will produce one output file: myapp.abs. The next step is to program an EPROM using this file.

Example – C and Ladder Logic Program in ROM

The C application program may share the ROM with a ladder logic program. The ladder logic program is always located at 100000h. The C program may start at any address that is a multiple of 100h following the ladder logic program.

The standard command file needs to be modified to link the application described in the previous examples. Copy the approm.cmd file to myapp.cmd.

Assume for this example that the ladder logic program ends at address 100417h. The next multiple of 100h after this address is 100500h. Modify the section locations to read:

```
; ----------------------------------------------
; Specify location and order of memory sections
;
; Note: the far_appcode section address must
;       be a multiple of 100h.
; ----------------------------------------------
sect far_appcode  = 00100500h

Modify the application object modules section to read:
; ----------------------------------------------
; Load application program object modules
; ----------------------------------------------
load main
load func1
load func2
```

Link the file with the command

lnkm77 -c myapp.cmd

This will produce one output file: myapp.abs. The next step is to program an EPROM using this file.

Controller Initialization

You should initialize the memory of the controller when beginning a new programming project or when you wish to start from default conditions. It is not necessary to initialize the controller before every program load.

To initialize the controller, first perform a SERVICE boot. A SERVICE boot preserves programs and data in nonvolatile RAM, but does not start the programs running. Default communication parameters are used.

To perform a service boot:

- Remove power from the controller.
- Press and hold the LED POWER switch.
- Apply power to the controller.
- Wait until the STAT LED on the top of the board turns on.
- Release the LED POWER switch.

Second, initialize the program and data memory in the controller. A new controller will require all initializations be performed. Selected initializations can be performed on a controller that is in use.

- Run the Telepace program under Microsoft Windows.
- Connect the PC to the controller with the appropriate serial cable.
- From the **Controller** menu, select under **Type** the controller type that is connected. A check mark appears beside the desired type when it is selected.
- From the **Controller** menu, select the **Initialize** command.
- Select all options: **Erase Ladder Logic Program, Erase C Program**, **Initialize Controller** and **Erase Register Assignment Table**.
- Click on the **OK** button.

Loading Programs into RAM

The *C Program Loader* dialog transfers executable files from a PC to the controller.

To load a program into RAM:

- Initialize the controller (see ***Controller Initialization*** section above).
- Load the program into the controller:
- Run the Telepace program.
- From the **Controller** menu, select the **C Program Loader** command.
- Enter the executable (.abs) file in the edit box for the C Program file name.
- Select the **C Program** write option and any other write options desired.

- Click on the **Write** button. The file will be downloaded.

A checksum is calculated for the complete C program. The checksum is verified each time the program is run. This prevents a damaged program from running.

Loading Programs into EPROM

The procedure for creating an EPROM depends on your EPROM programmer. In general you need to follow these steps:

- Load the executable file into the programmer and program the EPROM.
- Install the EPROM in the controller.

The controller can accept the following EPROMs. Other EPROMs may be compatible. Contact Control Microsystems if you are considering using an EPROM not in this list.

Size (Kbytes)	Manufacturer	Part Number
64	AMD	AM27C512-70DC
64	SGS-Thomson	M27C512-80F1
128	AMD	AM27C010-70DC
128	Atmel	AT27C010-70PI (one time programmable)
128	SGS-Thomson	M27C1001-80F1
128	Toshiba	TC57H1000AD-85
256	AMD	AM27C020-70DC
256	SGS-Thomson	M27C2001-80F1

C Programs may be loaded into Flash memory or EPROM when using Telepace firmware 1.64 or older.

Telepace firmware 1.65 or newer no longer supports C Programs in Flash memory. C Programs may be loaded in RAM memory only.

Creating the EPROM

Load the executable (.abs) file into the memory of the EPROM programmer, according to the instructions for the programmer.

The first byte of the EPROM (offset 0 in the EPROM) maps to address 100000h when the EPROM is installed in the controller. The linker generates an executable file with address offsets starting at 100000h. These offsets need to be removed with programmers, so that the memory image can be placed at offset 0 in the EPROM itself. (this does not affect the addresses in the program itself, just the address at which it loads.)

Consult your EPROM programmer documentation to determine how to remove the offset. This is typically done in one of two ways:

- Specify the data is to be loaded from file address 100000h. You may have to specify that the file is loaded to offset 0h.

- Or, specify a load offset of –100000h when reading the executable file. The programmer will add –100000h to all load addresses in the file, resulting in a memory image at offset 0h.

Program the EPROM according to the instructions for your programmer.

Installing the EPROM

Install the EPROM in the application ROM socket on the 5203 or 5204 controller board:

- Locate the socket labeled U14. This is the application ROM socket.
- Orient the EPROM so the notch on the EPROM is at the same end as the notch in the socket.
- Align all pins of the EPROM with the socket.
- Press the EPROM gently into the socket.
- Check that all pins are inserted correctly and that none are bent.

Initialize the controller (see ***Controller Initialization*** section above). The **Erase C Program** option needs to be specified. Other initializations may be performed if desired.

Executing Programs

C application programs are executed when a *run program* command is received from the Telepace C Program Loader; or power is applied to the controller (except when a SERVICE boot is performed).

To start a program from the program loader:

- Run the Telepace program.
- From the **C Program Loader** dialog, click on the **Run** button to execute the program.

The controller will execute either the program in RAM or the program in ROM. It chooses the program to execute in the following order:

- C application program in RAM;
- C application program in ROM;
- no C application (standard start-up sequence for other components).

This mode of operation is useful in the following scenario. A controller is installed with a program in ROM. If new features or corrections are required, a program can be downloaded into RAM, either locally or remotely. This program will take precedence over the program in ROM.

If the RAM program is lost or damaged, the ROM program will execute. The ROM program can be used as a fallback, performing minimal functions to maintain a process.

Real Time Operating System

The real time operating system (RTOS) provides the programmer with tools for building sophisticated applications. The RTOS allows pre-emptive scheduling of event driven tasks to provide quick response to real-world events. Tasks multi-task cooperatively. Inter-task communication and event notification functions pass information between tasks. Resource functions facilitate management of non-sharable resources.

Task Management

The task management functions provide for the creation and termination of tasks. Tasks are independently executing routines. The RTOS uses a cooperative multi-tasking scheme, with pre-emptive scheduling of event driven tasks.

The initial task (the **main** function) may create additional tasks. The RTOS supports up to 16 tasks. There are 5 task priority levels to aid in scheduling of task execution.

Task Execution

SCADAPack controllers can execute one task at a time. The RTOS switches between the tasks to provide parallel execution of multiple tasks. The application program can be event driven, or tasks can execute round-robin (one after another).

Task execution is based upon the priority of tasks. There are 5 priority levels. Level 0 is reserved for the null task. This task runs when there are no other tasks available for execution. Application programs can use levels 1 to 4. The main task is created at priority level 1.

Tasks that are not running are held in queues. The Ready Queue holds all tasks that are ready to run. Event queues hold tasks that are waiting for events. Message queues hold tasks waiting for messages. Resource queues hold tasks that are waiting for resources. The envelope queue holds tasks that are waiting for envelopes.

Priority Inversion Prevention

When a higher priority task, Task H, requests a resource, which is already obtained by a lower priority task, Task L, the higher priority task, is blocked until Task L releases the resource. If Task L is unable to execute to the point where its releases the resource, Task H will remain blocked. This is called a Priority Inversion.

To stop this from occurring, the prevention method known as Priority Inheritance has been implemented. In the example already described, the lower priority task, Task L, is promoted to the priority of Task H until it releases the needed resource. At this point Task L is returned to its original priority. Task H will obtain the resource now that it is available.

This does not stop deadlocks that occur when each task requests a resource that the other has already obtained. This “deadly embrace” is a design error in the application program.

Task Management Functions

There are five RTOS functions for task management. Refer to the ***Function Specification*** section for details on each function listed.

create_task	Create a task and make it ready to execute.
end_task	Terminate a task and free the resources and envelopes allocated to it.
end_application	Terminate all application program type tasks. This function is used by communication protocols to stop the application program prior to loading new code.
installExitHandler	Specify a function that is called when a task is ended with the end_task or end_application functions.
getTaskInfo	Return information about a task.

Task Management Macros

The **ctools.h** file defines the following macros used for task management. Refer to the ***C Tools Macros*** section for details on each macro listed.

RTOS_PRIORITIES	Number of RTOS task priorities.
RTOS_TASKS	Number of RTOS tasks.
STACK_SIZE	Size of the machine stack.
TS_EXECUTING	Task status indicating task is executing
TS_READY	Task status indicating task is ready to execute
TS_WAIT_RESOURCE	Task status indicating task is blocked waiting for a resource
TS_WAIT_ENVELOPE	Task status indicating task is blocked waiting for an envelope
TS_WAIT_EVENT	Task status indicating task is blocked waiting for an event
TS_WAIT_MESSAGE	Task status indicating task is blocked waiting for a message

Task Management Structures

The **ctools.h** file defines the structure **Task Information Structure** for task management information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Resource Management

The resource management functions arbitrate access to non-sharable resources. These resources include physical devices such as serial ports, and software that is not re-entrant.

The RTOS defines nine system resources, which are used by components of the I/O drivers, memory allocation functions and communication protocols.

An application program may define other resources as required. Care needs to be taken not to duplicate any of the resource numbers declared in **ctools.h** as system resources.

Resource Management Functions

There are three RTOS functions for resource management. Refer to the ***Function Specification*** section for details on each function listed.

request_resource	Request access to a resource and wait if the resource is not available.
poll_resource	Request access to a resource. Continue execution if the resource is not available
release_resource	Free a resource for use by other tasks.

IO_SYSTEM Resource

The IO_SYSTEM resource regulates access to all functions using the I/O system. C application programs, ladder logic programs, communication protocols and background I/O operations share the I/O system. It is imperative the resource is obtained to prevent a conflict, as protocols and background operations are interrupt driven. Retaining control of the resource for more that 0.1 seconds will cause background operations will to not execute properly.

DYNAMIC_MEMORY Resource

The DYNAMIC_MEMORY resource regulates access to all memory allocation functions. These functions allocate memory from the system heap. The heap is shared amongst all tasks. The allocation functions are non-reentrant.

The DYNAMIC_MEMORY resource needs to be obtained before using any of the following functions.

calloc	allocates data space dynamically
free	frees dynamically allocated memory
malloc	allocates data space dynamically
realloc	changes the size of dynamically allocated space

AB_PARSER Resource

This resource is used by the DF1 communication protocol tasks to allocate access to the common message parser for each serial port. This resource is of no interest to an application program. However, an application program may not use the resource number assigned to it.

MODBUS_PARSER Resource

This resource is used by Modbus communication protocol drivers to allocate access to the common message parser by tasks for each serial port. This resource is of no interest to an application program.

Resource Management Macros

The **ctools.h** file defines the following macros used for resource management. Refer to the ***C Tools Macros*** section for details on each macro listed.

AB_PARSER	DF1 protocol message parser.
COM1_DIALUP	Resource for dialing functions on com1.
COM2_DIALUP	Resource for dialing functions on com2.
COM3_DIALUP	Resource for dialing functions on com3.
COM4_DIALUP	Resource for dialing functions on com4.
DYNAMIC_MEMORY	Memory allocation functions.
HART	HART modem resource.
IO_SYSTEM	I/O system hardware functions.
MODBUS_PARSER	Modbus protocol message parser.
RTOS_RESOURCES	Number of RTOS resource flags.

Inter-task Communication

The inter-task communication functions pass information between tasks. These functions can be used for data exchange and task synchronization. Messages are queued by the RTOS until the receiving task is ready to process the data.

Inter-task Communication Functions

There are five RTOS functions for inter-task communication. Refer to the ***Function Specification*** section for details on each function listed.

send_message	Send a message envelope to another task.
receive_message	Read a received message from the task's message queue or wait if the queue is empty.
poll_message	Read a received message from the task's message queue. Continue execution of the task if the queue is empty.
allocate_envelope	Obtain a message envelope from free pool maintained by the RTOS, or wait if none is available.
deallocate_envelope	Return a message envelope to the free pool maintained by the RTOS.

Inter-task Communication Macros

The **ctools.h** file defines the following macros used for inter-task communication. Refer to the ***C Tools Macros*** section for details on each macro listed.

MSG_DATA Specifies the data field in an envelope contains a data value.

MSG_POINTER Specifies the data field in an envelope contains a pointer.

RTOS_ENVELOPES Number of RTOS envelopes.

Inter-task Communication Structures

The **ctools.h** file defines the structure **Message Envelope Structure** for inter-task communication information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Event Notification

The event notification functions provide a mechanism for communicating the occurrence events without specifying the task that will act upon the event. This is different from inter-task communication, which communicates to a specific task.

Multiple occurrences of a single type of event are queued by the RTOS until a task waits for or polls the event.

Event Notification Functions

There are four RTOS functions for event notification. Refer to the ***Function Specification*** section for details on each function listed.

wait_event Wait for an event to occur.

poll_event Check if an event has occurred. Continue execution if one has not occurred.

signal_event Signal that an event has occurred.

interrupt_signal_event Signal that an event has occurred from an interrupt handler. This function must only be called from within an interrupt handler.

There are two support functions, which are not part of the RTOS that may be used with events.

startTimedEvent Enables signaling of an event at regular intervals.

endTimedEvent Terminates signaling of a regular event.

Event Notification Macros

The **ctools.h** file defines the following macro used for event notification. Refer to the ***C Tools Macros*** section for details.

RTOS_EVENTS Defines the number of available RTOS events.

System Events

The RTOS defines events for communication port management and background I/O operations. An application program may define other events as required. Take care not to duplicate any of the event numbers declared in **ctools.h** as system events.

BACKGROUND	This event triggers execution of the background I/O routines. An application program cannot use it.
COM1_RCVR	This event is used by communication protocols to signal a character or message received on com1. It can be used in a custom character handler (see **install_handler**).
COM2_RCVR	This event is used by communication protocols to signal a character or message received on com2. It can be used in a custom character handler (see **install_handler**).
COM3_RCVR	This event is used by communication protocols to signal a character or message received on com3. It can be used in a custom character handler (see **install_handler**).
COM4_RCVR	This event is used by communication protocols to signal a character or message received on com4. It can be used in a custom character handler (see **install_handler**).
FOXCOM_MSG_RECEIVED	This event is used when a Foxcom message is received. An application program cannot use it.
FOXCOM_STARTED	This event is used when Foxcom communication has been established with the sensor. An application program cannot use it.
NEVER	This event never occurs. It can be used to disable a task by waiting for it to occur. However, to end a task it is better to use **end_task**. This frees all resources and stack space allocated to the task.

Error Reporting

Sharable I/O drivers to return error information to the calling task use the error reporting functions. These functions ensure that an error code generated by one task is not reported in another task. The **errno** global variable used by some functions may be modified by another task, before the current task can read it.

Error Reporting Functions

There are two RTOS functions for error reporting. Refer to the ***Function Specification*** section for details on each function listed.

check_error	Check the error code for the current task.

report_error Set the error code for the current task.

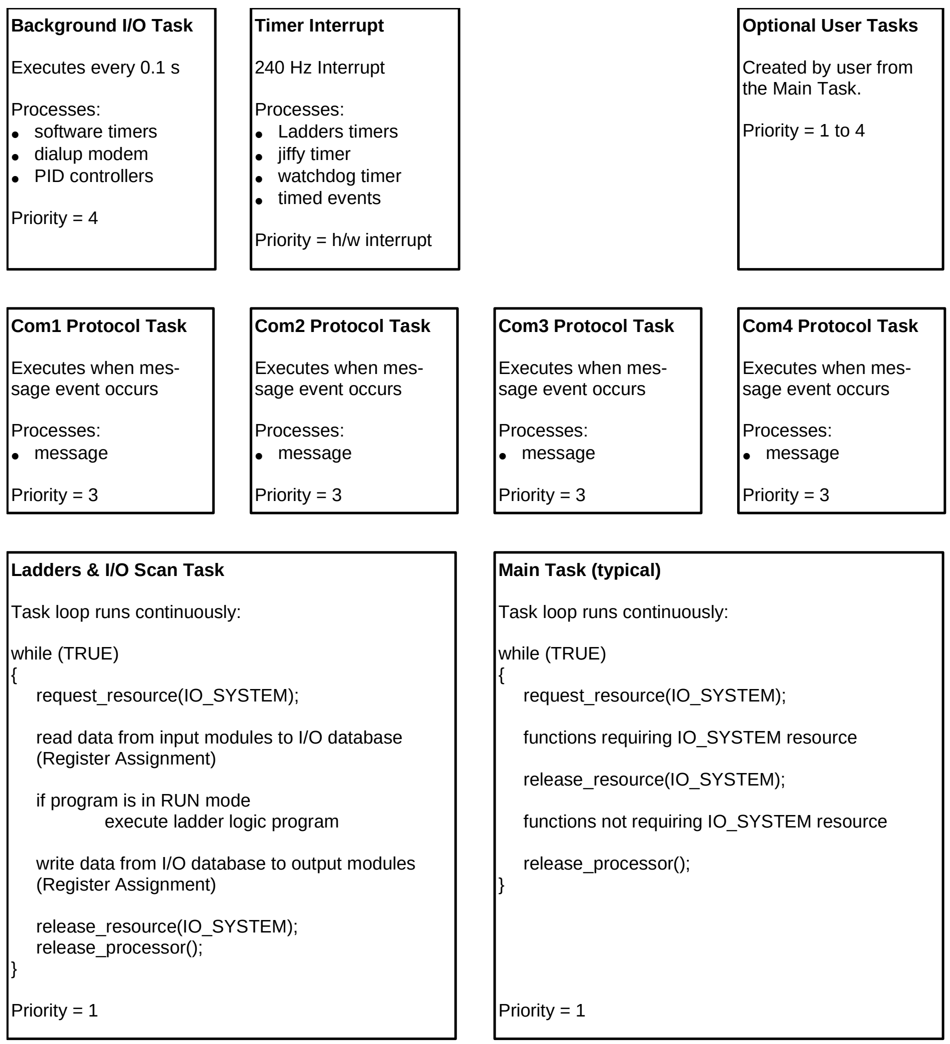
Error Reporting Macros

The **ctools.h** file defines the following macro used for error reporting. Refer to the C Tools Macros section for details.

NO_ERROR Error code indicating no error has occurred.

SCADAPack Task Architecture

The diagram shows the tasks present in the SCADAPack controller.



The highest priority routines that execute are hardware interrupt handlers. Most hardware interrupt handlers perform their functions transparently. The Timer Interrupt handler is important to application programs, because it updates several timers that can be used in application programs. It also triggers the background I/O task.

The background I/O task is the highest priority task in the system. It processes software timers, PID controllers and dialup modem control routines.

There is one protocol task for each serial port where a protocol is enabled. The protocol tasks wait for an event signaled by an interrupt handler. This event is signaled when a complete message is received. The protocol tasks process the received message and transmit a response when needed. Protocol tasks may be disabled and replaced with protocol tasks from the application program.

The Ladder Logic and I/O Scan task executes the Ladder Logic program and performs an I/O scan based on the register assignment. This task is the same priority as the main user application task.

The main task is the central task of the user application. It performs the functions required by the user. Typically, it executes at the same priority as the Ladder Logic and I/O Scan task. It may start other user tasks if needed.

RTOS Example Application Program

The following program is used in the explanation of the RTOS functions. It creates several simple tasks that demonstrate how tasks execute. A task is a C language function that has as its body an infinite loop so it continues to execute forever.

The main task creates two tasks. The echoData task is higher priority than main. The auxiliary task is the same priority as main. The main task then executes round robin with other tasks of the same priority.

The auxiliary task is a simple task that executes round robin with the other tasks of its priority. Only the code necessary for task switching is shown to simplify the example.

The echoData task waits for a character to be received on a serial port, then echoes it back out the port. It waits for the event of the character being received to allow lower priority tasks to execute. It installs a character handler function – signalCharacter – that signals an event each time a character is received. This function is hooked into the receiver interrupt handler for the serial port.

The execution of this program is explained in the ***Explanation of Task Execution*** section.

```
/* -----------------------------------------------------------------
   SCADAPack Real Time Operating System Sample
   Copyright (c) 1998, Control Microsystems Inc.

   Version History
       version 1.00  Wayne Johnston      November 10, 1998
   ----------------------------------------------------------------
*/

/* ---- Version 1.00 ---------------------------------------------
```

```
   This program creates several simple tasks for demonstration of
the
   functionality of the real time operation system.
   ---------------------------------------------------------------
*/

#include <mriext.h>
#include <stdio.h>
#include "ctools,h"

/* ----------------------------------------------------------------
   Constants
   ---------------------------------------------------------------
*/

#define CHARACTER_RECEIVED 10

/* ----------------------------------------------------------------
   signalCharacter

   The signalCharacter function signals an event when a character
is
   received. This function must be called from an interrupt
handler.
   ----------------------------------------------------------------
*/

void signalCharacter(unsigned character, unsigned error)
{
       /* If there was no error, signal that a character was
received */
       if (error == 0)
              {
              interrupt_signal_event(CHARACTER_RECEIVED);
              }

       /* Prevent compiler unused variables warning (generates no
code) */
       character;
}

/* ---------------------------------------------------------------
   echoData

   The echoData function is a task that waits for a character
   to be received on com6 and echoes the character back. It
installs
   a character handler for com6 to generate events on the
reception
   of characters.
   ----------------------------------------------------------------
*/
```

3

```
void echoData(void)
```

```
{
       struct prot_settings protocolSettings;
       struct pconfig portSettings;
       int character;

       /* Disable communication protocol */
       get_protocol(com6, &protocolSettings);
       protocolSettings.type = NO_PROTOCOL;
       set_protocol(com6, &protocolSettings);

       /* Set serial communication parameters */
       portSettings.baud      = BAUD9600;
       portSettings.duplex    = FULL;
       portSettings.parity    = NONE;
       portSettings.data_bits = DATA8;
       portSettings.stop_bits = STOP1;
       portSettings.flow_rx   = DISABLE;
       portSettings.flow_tx   = DISABLE;
       portSettings.type      = RS232;
       portSettings.timeout   = 600;
       set_port(com6, &portSettings);

       /* Install handler for received character */
       install_handler(com6, signalCharacter);

       while (TRUE)
              {
              /* Wait for a character to be received */
<--------- 4 9
              wait_event(CHARACTER_RECEIVED);

<------------ 8
              /* Echo the character back */
              character = fgetc(com6);
              fputc(character, com6);
              }
}


/* ----------------------------------------------------------------
-----
   auxiliary

   The auxiliary function is a task that performs some action
   required by the program. It does not have specific function so
   that the real time operating system features are clearer.
   -------------------------------------------------------------
----- */

void auxiliary(void)
<------------ 7
{
       while (TRUE)
```

```
        {
                /* ... add application specific code here ... */

                /* Allow other tasks of this priority to run */
                release_processor();
                }
}

/* -----------------------------------------------------------------
-----
   main

   This function creates two tasks: one at priority three and one
at
   priority 1 to demonstrate the functions of the RTOS.
   -----------------------------------------------------------------
----- */
```

1

```
void main(void)
```

2

```
{
        /* Create serial communication task */
        create_task(echoData, 3, APPLICATION, 3);

        /* Create a task - same priority as main() task */
        create_task(auxiliary, 1, APPLICATION, 2);
```

5

```
        while (TRUE)
                {
                /* ... add application specific code here ... */

                /* Allow other tasks of this priority to execute */
```

6

```
                release_processor();
                }
}
```

Explanation of Task Execution

SCADAPack controllers can execute one task at a time. The Real Time Operating System (RTOS) switches between the tasks to provide parallel execution of multiple tasks. The application program can be event driven, or tasks can execute round-robin (one after another). This program illustrates both types of execution.

Task execution is based upon the priority of tasks. There are 5 priority levels. Level 0 is reserved for the null task. This task runs when there are no other tasks available for execution. Application programs can use levels 1 to 4. The main task is created at priority level 1.

Tasks that are not running are held in queues. The Ready Queue holds all tasks that are ready to run. Event queues hold tasks that are waiting for events. Message queues hold tasks waiting for messages. Resource queues hold tasks that are waiting for resources. The envelope queue holds tasks that are waiting for envelopes.

The execution of the tasks is illustrated by examining the state of the queues at various points in the program. These points are indicated on the program listing above. The examples show only the Ready queue, the Event 10 queue and the executing task. These are the only queues relevant to the example.

Execution Point 1

This point occurs just before the main task begins. The main task has not been created by the RTOS. The null task has been created, but is not running. No task is executing.

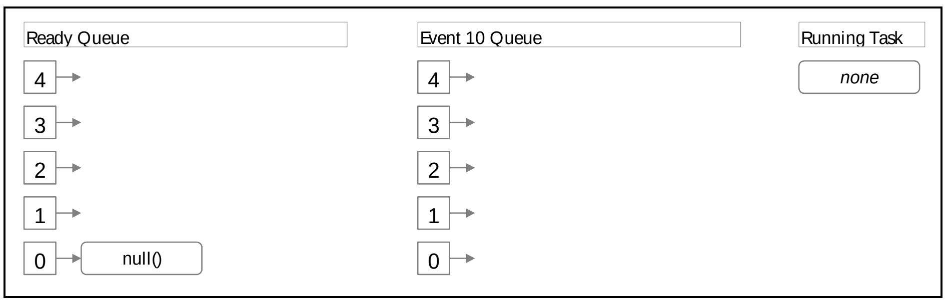


Figure 1: Queue Status before Execution of main Task

Execution Point 2

This point occurs just after the creation of the main task. It is the running task. On the next instruction it will create the echoData task.

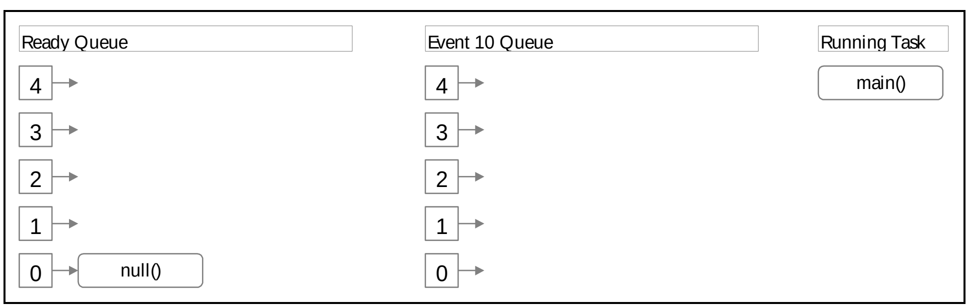


Figure 2: Queue Status at Start of main Task

Execution Point 3

This point occurs just after the echoData task is created. The echoData task is higher priority than the main task so it is made the running task. The main task is placed into the ready queue. It will execute when it becomes the highest priority task.

The echoData task initializes the serial port and installs the serial port handler function signalCharacter. It will then wait for an event. This will suspend the task until the event occurs.

The signalCharacter function will generate an event each time a character is received without an error.

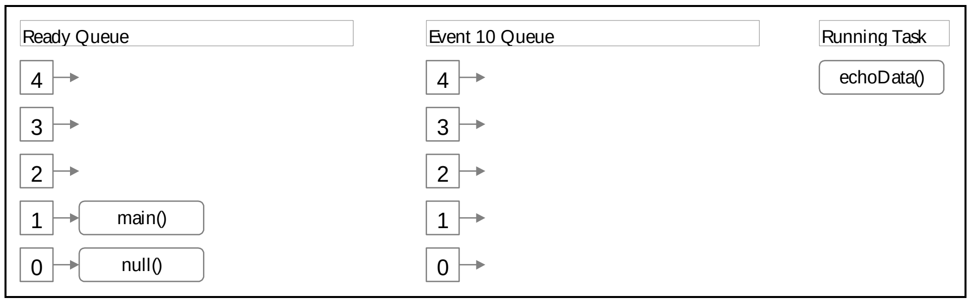


Figure 3: Queue Status after Creation of echoData Task

Execution Point 4

This point occurs just after the echoData task waits for event 10. It has been placed on the event queue for event 10.

The highest priority task on the ready queue was the main task. It is now running. On the next instruction it will create another task at the same priority as main.

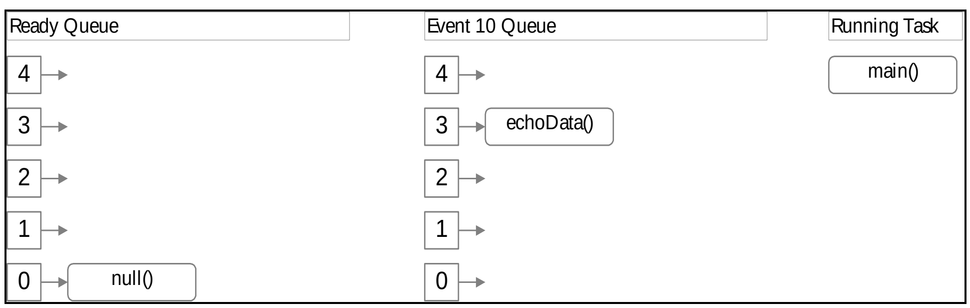


Figure 4: Queue Status After echoData Task Waits for Event

Execution Point 5

This point occurs just after the creation of the **auxiliary** task. This task is the same priority as the **main** task. Therefore the **main** task remains the running task. The **auxiliary** task is ready to run and it is placed on the Ready queue.

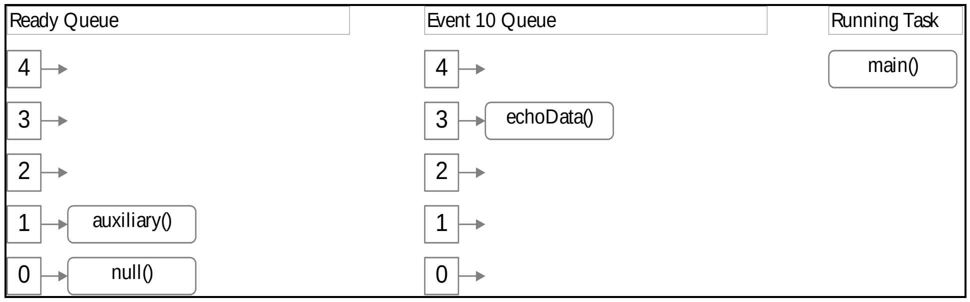


Figure 5 Queue Status after Creation of auxiliary Task

Execution Point 6

This point occurs just after the main task releases the processor, but before the next task is selected to run. The main task is added to the end of the priority 1 list in the Ready queue.

On the next instruction the RTOS will select the highest priority task in the Ready queue.

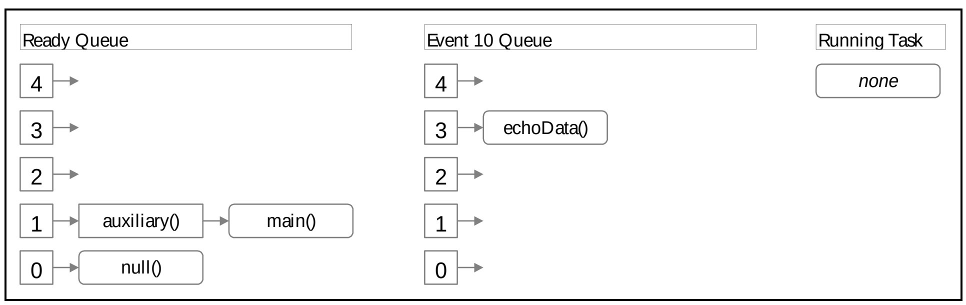


Figure 6: Queue Status After main Task Releases Processor

Execution Point 7

This point is just after the auxiliary task has started to run. The main and auxiliary tasks will continue to alternate execution, as each task releases the processor to the other.

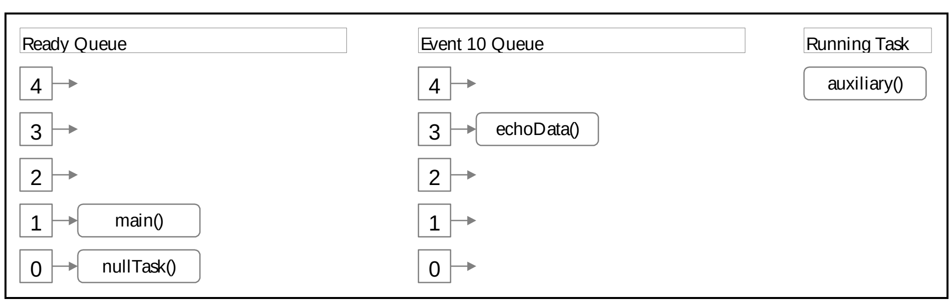


Figure 7: Queue Status at Start of auxiliary Task

Execution Point 8

This point occurs just after a character has been received. The signalCharacter function executes and signals an event. The RTOS checks the event queue for the event, and makes the highest priority task ready to execute. In this case the echoData task is made ready.

The RTOS then determines if the new task is higher priority than the executing task. Since the echoData task is higher priority than the auxiliary task, a task switch occurs. The auxiliary task is placed on the Ready queue. The echoData task executes.

Observe the position of auxiliary in the Ready queue. The main task will execute before it at the next task switch.

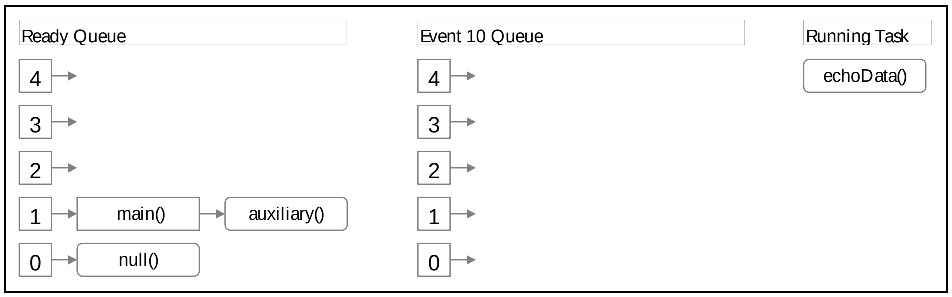


Figure 8: Queue Status after Character Received

Execution Point 9

This point occurs just after the echoData task waits for the character-received event. It is placed on the event 10 queue. The highest priority task on the ready queue – main – is given the processor and executes.

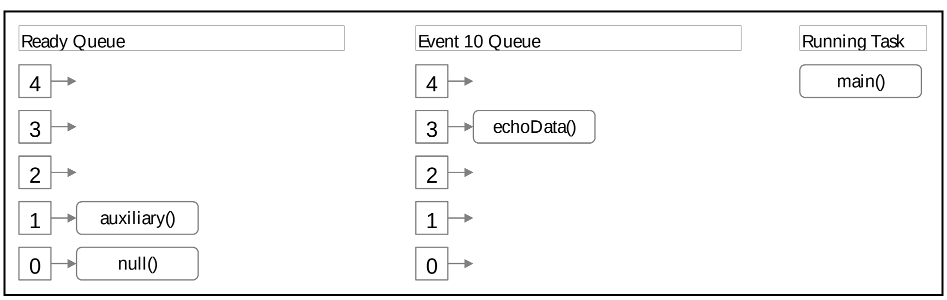


Figure 9: Queue Status after echoData Waits for Event

Overview of Programming Functions

This section of the User Manual provides and overview of the Functions, Macros, Structure and Types available to the user. The Functions, Macros, Structure and Types overview is separated into sections of related functions. Refer to the Function Specification, C Tools Macros and C Tools Structures and Types section of this manual for detailed explanations of the Functions, Macros, Structure and Types described here.

Controller Operation

This section of the manual provides an overview of the Telepace functions relating to controller operation. These functions are provided in addition to the run-time library supplied with the Microtec C compiler.

Start Up Functions

There are two library functions related to the system or application start up task. Refer to the ***Function Specification*** section for details on each function listed.

startup_task	Returns the address of the system start up routine.
system_start	The default start up routine.

Start Up Macros

The **ctools.h** file defines the following macros for use with the start up task. Refer to the ***C Tools Macros*** section for details on each macro listed.

STARTUP_APPLICATION	Specifies the application start up task.
STARTUP_SYSTEM	Specifies the system start up task.

Start Up Task Info Structure

The **ctools.h** file defines the structure **Start Up Information Structure** for use with the startup_task function. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Program Status Information Functions

There are five library functions related to controller program status information. Refer to the ***Function Specification*** section for details on each function listed.

applicationChecksum	Returns the application program checksum.
getBootType	Returns the controller boot up status.
getProgramStatus	Returns the application program execution status.
setBootType	Sets the controller boot up status.

setProgramStatus	Sets the application program execution status.

Program Status Information Macros

The **ctools.h** file defines the following macros for use with controller program information. Refer to the ***C Tools Macros*** section for details on each macro listed.

NEW_PROGRAM	Application program is newly loaded.
PROGRAM_EXECUTED	Application program has been executed.
COLD_BOOT	Controller started in COLD BOOT mode.
RUN	Controller started in RUN mode.
SERVICE	Controller started in SERVICE mode.
REENTRY_BOOT	

Controller Information Functions

There is one library function related to controller information. Refer to the ***Function Specification*** section for details on the function listed.

getControllerID	Returns the controller ID string.

Controller Information Macros

The **ctools.h** file defines the following macros for use with controller information. Refer to the ***Function Specification*** section for details on each macro listed.

AB_PROTOCOL	DF1 protocol firmware option
BASE_TYPE_MASK	Controller type bit mask
FT_NONE	Unknown firmware type
FT_TELEPACE	Telepace firmware type
FT_ISAGRAF	IEC 61131-3 firmware type
GASFLOW	Gas Flow calculation firmware option
RUNS_2	Set if Gas Flow supports two meter runs
SCADAPACK	SCADAPack controller
SCADAPACK_LIGHT	SCADAPack LIGHT controller
SCADAPACK_PLUS	SCADAPack PLUS controller
UNKNOWN_CONTOLLER	Unknown controller type

Firmware Version Information Functions

There is one function related to the controller firmware version. Refer to the ***Function Specification*** section for details.

getVersion	Returns controller firmware version information.

Firmware Version Information Macros

The **ctools.h** file defines the following macros for use with the firmware version function. Refer to the ***C Tools Macros*** section for details on each macro listed.

VI_DATE_SIZE	Number of characters in the version information date field.
VI_STRING_SIZE	Number of characters in the version information copyright field.

Firmware Version Information Structure

The **ctools.h** file defines the structure **Version Information Structure** for controller firmware version information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Sleep Mode Functions

SCADAPack controllers are capable of extremely low power operation when in sleep mode. SCADAPack controllers enter the sleep mode under control of the application program. Refer to the ***SCADAPack System Hardware Manual*** for further information on controller sleep mode.

There are three library functions related to sleep mode. Refer to the ***Function Specification*** section for details on each function listed.

getWakeSource	Gets wake up sources
setWakeSource	Sets wake up sources
sleep	Put controller into sleep mode

Sleep Mode Macros

The **ctools.h** file defines the following macros for use sleep mode. Refer to the ***C Tools Macros*** section for details on each macro listed.

SLEEP_MODE_SUPPORTED	Defined if sleep function is supported
WS_ALL	All wake up sources enabled
WS_COUNTER_0_OVERFLOW	Bit mask to enable counter 0 overflow as wake up source
WS_COUNTER_1_OVERFLOW	Bit mask to enable counter 1 overflow as wake up source
WS_COUNTER_2_OVERFLOW	Bit mask to enable counter 2 overflow as wake up source
WS_INTERRUPT_INPUT	Bit mask to enable interrupt input as wake up source
WS_LED_POWER_SWITCH	Bit mask to enable LED power switch as wake up source
WS_NONE	No wake up source enabled

WS_REAL_TIME_CLOCK	Bit mask to enable real time clock as wake up source
WS_UNDEFINED	Undefined wake up source

Power Management Functions

Under normal operation, the SCADAPack 350 operates on a CPU clock frequency of 32 MHz. However, the SCADAPack 350 controller is capable of operating on a reduced CPU clock frequency of 8 MHz, known as Reduced Power Mode.

Further power savings can be realized on the SCADAPack 350 controller by disabling the LAN or USB peripheral and host ports. Activation of Reduced Power mode as well as the deactivation of the communication ports can be performed by the application program.

The library functions associated with the aforementioned power management allows for the following:

- The CPU speed can be changed from full speed (32 MHz) to reduced speed (8 MHz).
- The LAN port can be enabled or disabled
- The USB peripheral port can be enabled or disabled
- The USB host port can be enabled or disabled.

The Power Mode LED blinks once a second when the controller is operating in Reduced Power Mode.

The library functions associated with the power management features are listed below. Refer to the ***Function Specification*** section for details on each function listed.

getPowerMode	Gets the current power mode
setPowerMode	Sets the power mode

Power Management Macros

The **ctools.h** file defines the following macros for use in the power management functions. Refer to the ***C Tools Macros*** section for details on each macro listed.

PM_CPU_FULL	The CPU is set to run at full speed
PM_CPU_REDUCED	The CPU is set to run at a reduced speed
PM_CPU_SLEEP	The CPU is set to sleep mode
PM_LAN_ENABLED	The LAN is enabled
PM_LAN_DISABLED	The LAN is disabled
PM_USB_PERIPHERAL_ENABLED	The USB peripheral port is enabled
PM_USB_PERIPHERAL_DISABLED	The USB peripheral port is disabled
PM_USB_HOST_ENABLED	The USB host port is enabled

PM_USB_HOST_DISABLED The USB host port is disabled

PM_UNAVAILABLE The status of the device could not be read

Configuration Data EEPROM Functions

The EEPROM is nonvolatile memory used to store configuration parameters. The application program cannot store application data into this memory. It can cause the system configuration parameters to be written, using the **save** function.

The contents of the EEPROM are copied to RAM under two conditions: during a RUN boot of the controller; and when the application program executes the **load** function.

The following data is loaded on a RUN boot; otherwise default information is used:

- serial port configuration tables
- protocol configuration tables
- enable store and forward settings
- LED power settings
- mask for wake-up sources
- execution period on power-up for each PID

There are two library functions related to the configuration data EEPROM. Refer to the ***Function Specification*** section for details on each function listed.

Save Writes configuration data from RAM to EEPROM

load Reads configuration data from EEPROM into RAM

Configuration Data EEPROM Macros

The **ctools.h** file defines the following macros for use with the configuration data EEPROM. Refer to the ***C Tools Macros*** section for details on each macro listed.

EEPROM_EVERY EEPROM section loaded to RAM on every CPU reboot.

EEPROM_RUN EEPROM section loaded to RAM on RUN type boots only.

EEPROM_SUPPORTED If defined, indicates that there is an EEPROM in the controller.

I/O Bus Communication Functions

The **ctools.h** file defines the following functions that access the I/O bus. The I/O bus is I^2C compatible. Refer to the ***Function Specification*** section for details on each function listed.

ioBusReadByte Reads one byte from an I^2C slave device

ioBusReadLastByte	Reads one byte from an I^2C slave device and terminates read
ioBusReadMessage	Reads a message from an I^2C slave device
ioBusSelectForRead	Selects an I^2C slave device for reading
ioBusSelectForWrite	Selects an I^2C slave device for writing
ioBusStart	Issues an I^2C bus START condition
ioBusStop	Issues an I^2C bus STOP condition
ioBusWriteByte	Writes one byte to an I^2C slave device
ioBusWriteMessage	Writes a message to an I^2C slave device

I/O Bus Communication Macros

The **ctools.h** file defines the following macros for use with I/O Bus Communication. Refer to the ***C Tools Macros*** section for details on each macro listed.

The **ctools.h** file defines the following macros.

READSTATUS	enumeration type ReadStatus
WRITESTATUS	enumeration type WriteStatus

I/O Bus Communication Types

The **ctools.h** file defines the enumeration types **ReadStatus** and **WriteStatus**. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

System Functions

The **ctools.h** file defines the following functions for system initialization and for retrieving system information. Some of these functions are primarily used in the **appstart.c** routine, having limited use in an application program.

Refer to the ***Function Specification*** section for details on each function listed.

applicationChecksum	Returns the application program checksum.
ioClear	Clears all I/O points
ioDatabaseReset	Resets the controller to default settings.
ioRefresh	Refresh outputs with internal data
ioReset	Reset all I/O modules

Controller I/O Hardware

This section of the manual provides an overview of the Telepace C Tools functions relating to controller signal input and output (I/O). These functions are provided in addition to the run-time library supplied with the Microtec C compiler.

Analog Input Functions

The controller supports internal analog inputs and external analog input modules. Refer to the ***SCADAPack System Hardware Manual*** for further information on controller analog inputs and analog input modules.

There are several library functions related to internal analog inputs and analog input modules. Refer to the ***Function Specification*** section for details on each function listed.

readBattery	Read the controller RAM battery voltage.
readThermistor	Read the controller ambient temperature sensor.
readInternalAD	Read the controller internal AD converter.
ioRead4Ain	Read 4 analog inputs into I/O database.
ioRead8Ain	**R**ead 8 analog inputs into I/O database.
IoRead4202Inputs	Read the digital, counter and analog inputs from a SCADAPack 4202 DR.
IoRead4202DSInputs	Read the digital, counter and analog inputs from a SCADAPack 4202 DS.
ioRead5505Inputs	Read the digital and analog inputs from a 5505 I/O Module.
ioRead5506Inputs	Read the digital and analog inputs from a 5506 I/O Module.
ioRead5601Inputs	Read the digital and analog inputs from a SCADAPack 5601 I/O Module.
ioRead5602Inputs	Read the digital and analog inputs from a SCADAPack 5602 I/O Module.
ioRead5604Inputs	Read the digital and analog inputs from a SCADAPack 5604 I/O Module.
ioRead5606Inputs	Read the digital and analog inputs from a 5606 I/O Module.
ioReadLPInputs	Read the digital and analog inputs from the SCADAPack LP I/O.
ioReadSP100Inputs	Read the digital and analog inputs from the SCADAPack 100 I/O.

Analog Input Macros

The **ctools.h** file defines the following macros for use with controller analog inputs. Refer to the ***C Tools Macros*** section for details on each macro listed.

AD_BATTERY	Internal AD channel connected to lithium battery.
AD_THERMISTOR	Internal AD channel connected to thermistor.
T_CELSIUS	Specifies temperatures in degrees Celsius.

T_FAHRENHEIT	Specifies temperatures in degrees Fahrenheit.
T_KELVIN	Specifies temperatures in degrees Kelvin.
T_RANKINE	Specifies temperatures in degrees Rankine.

Analog Output Functions

The controller supports external analog output modules. Refer to the ***SCADAPack System Hardware Manual*** for further information on these modules.

There are three library functions related to analog output modules. Refer to the ***Function Specification*** section for details on each function listed.

ioWriteAout	Write to 4 analog outputs from I/O database.
ioWrite2Aout	Write to 2 analog outputs from I/O database.
ioWrite4Aout	Write to 4 analog outputs from I/O database.
IoWrite4202Outputs	Write to the analog outputs of a SCADAPack 4202 DR.
IoWrite4202OutputsEx	Write to analog outputs of a SCADAPack 4202 DR with extended IO (4202 DR with a digital output).
ioWrite5303Aout	Write to analog outputs of the 5303 module from I/O database**.**
ioWrite5606Outputs	Write to the digital and analog outputs of 5606 I/O Module.
ioWriteLPOutputs	Writes data to the digital and analog outputs of the SCADAPack LP I/O.

Digital Input Functions

The controller supports internal digital inputs and external digital input modules. Refer to the ***SCADAPack System Hardware Manual*** for further information on controller digital inputs and digital input modules.

There are several library functions related to digital inputs and external digital input modules. Refer to the ***Function Specification*** section for details on each function listed.

interruptInput	Read the controller interrupt input.
readCounterInput	Read the status of the counter input points on the controller board.
ioRead8Din	read 8 digital inputs into I/O database.
ioRead16Din	read 16 digital inputs into I/O database.
IoRead32Din	read 32 digital inputs into I/O database.
IoRead4202Inputs	Read the digital, counter and analog inputs from a SCADAPack 4202 DR.

IoRead4202DSInputs	Read the digital, counter and analog inputs from a SCADAPack 4202 DS.
ioRead5505Inputs	Read the digital and analog inputs from a 5505 I/O Module.
ioRead5506Inputs	Read the digital and analog inputs from a 5506 I/O Module.
ioRead5601Inputs	Read the digital and analog inputs from a 5601 I/O Module.
ioRead5602Inputs	Read the digital or analog inputs from a 5602 I/O Module.
ioRead5604Inputs	Read the digital and analog inputs from a SCADAPack 5604 I/O Module.
ioRead5606Inputs	Read the digital and analog inputs from a 5606 I/O Module.
ioReadLPInputs	Read the digital and analog inputs from the SCADAPack LP I/O.
ioReadSP100Inputs	Read the digital and analog inputs from the SCADAPack 100 I/O.

Digital Output Functions

The controller supports external digital output modules. Refer to the ***SCADAPack System Hardware Manual*** for further information on controller digital output modules.

There are several library functions related to digital output modules. Refer to the ***Function Specification*** section for details on each function listed.

interruptInput	Read the controller interrupt input.
ioWrite16Dout	Write data to any 16 point Digital output module.
IoWrite32Dout	Write data to any 32 point Digital output module.
IoWrite4202OutputsEx	Write to digital and analog outputs of the SCADAPack 4202 DR with extended IO (with digital output) from I/O database.
IoWrite4202DSOutputs	Write to digital outputs of the SCADAPack 4202 DS from I/O database**.**
ioWrite5601Outputs	Write to the digital and analog outputs of SCADAPack 5601 I/O Module.
ioWrite5602Outputs	Write to the digital and analog outputs of SCADAPack 5602 I/O Module.
ioWrite5604Outputs	Write to the digital and analog outputs of SCADAPack 5604 I/O Module.

ioWrite5606Outputs	Write to the digital and analog outputs of 5606 I/O Module.
ioWrite8Dout	Write data to any 8 point Digital output module.
ioWriteLPOutputs	Writes data to the digital and analog outputs of the SCADAPack LP I/O.
ioWriteSP100utputs	Writes data to the digital outputs of the SCADAPack 100 I/O.

Counter Input Functions

The controller supports internal counters and external counter modules. The counter registers are 32 bits, for a maximum count of 4,294,967,295. They roll over to 0 on the next count. The counter inputs measure the number of rising inputs. Refer to the ***SCADAPack System Hardware Manual*** for further information on controller counter inputs and counter input modules.

There are four library functions related to counters. Refer to the ***Function Specification*** section for details on each function listed.

readCounter	Read a SCADAPack, SCADAPack LP or SCADAPack 100 counter with or without automatic clearing of the counter register.
interruptCounter	Read the SCADAPack or SCADAPack LP interrupt input as a counter with or without automatic clearing of the counter value.
ioRead4Counter	Read any 4 point Counter input module.
IoRead4202Inputs	Read the digital, counter and analog inputs from a SCADAPack 4202 DR.
IoRead4202DSInputs	Read the digital, counter and analog inputs from a SCADAense 4202 DS.

Counter Input Macros

The **ctools.h** file defines the following macro for use with counter inputs. Refer to the ***C Tools Macros*** section for details.

LOCAL_COUNTERS	Number of controller counter inputs.

Status LED and Output Functions

The status LED and output indicate alarm conditions. The STAT LED blinks and the STATUS output opens when an alarm occurs. The STAT LED turns off and the STATUS output closes when all alarms clear.

The STAT LED blinks a binary sequence indicating alarm codes. The sequences consist of long and short flashes, followed by an off delay of 1 second. The sequence then repeats. The sequence may be read as the Controller Status Code.

Refer to the ***SCADAPack System Hardware Manual*** for further information on the status LED and digital output. There is no status output on the SCADAPack programmable controllers.

There are two library functions related to the status LED and digital output. Refer to the ***Function Specification*** section for details on each function listed.

clearStatusBit	Clears bits in controller status code.
setStatusBit	Sets the bits in controller status code.

Status LED and Output Macros

The **ctools.h** file defines the following macros for use with the status LED and digital output. Refer to the ***C Tools Macros*** section for details on each macro listed.

S_MODULE_FAILURE	Status LED code for I/O module communication failure
S_NORMAL	Status LED code for normal status

Options Switches Functions

The controller has three option switches located under the cover of the controller module. These switches are labeled OPTION 1,2 and 3. The option switches are user defined except when a SCADAPack I/O module or SCADAPack AOUT module used. In this case option switches 1 and 2 select the analog ranges. Refer to the ***SCADAPack System Hardware Manual*** for further information on option switches.

There are no option switches on the SCADAPack 100, SCADAPack LP or the SCADAPack 4000 programmable controllers.

There is one library function related to the controller option switches. Refer to the ***Function Specification*** section for details.

optionSwitch	Read option switch states.

Option Switches Macros

The **ctools.h** file defines the following macros for use with option switches. Refer to the ***C Tools Macros*** section for details on each macro listed.

CLOSED	Specifies switch is in closed position
OPEN	Specifies switch is in open position

LED Indicators Functions

An application program can control three LED indicators.

The RUN LED indicates the execution status of the program. The LED can be on or off. It remains in the last state until changed.

The STAT LED indicates error conditions. It outputs an error code as a binary sequence. The sequence repeats until a new error code is output. If the error code is zero, the status LED turns off.

The FORCE LED indicates locked I/O variables. Use this function with caution in application programs.

There are three library functions related to the LED indicators. Refer to the ***Function Specification*** section for details on each function listed.

runLed	Controls the RUN LED status.
setStatus	Sets controller status code.
forceLed	Sets state of the force LED.

LED Indicators Macros

The **ctools.h** file defines the following macros for use with LED power control. Refer to the ***C Tools Macros*** section for details on each macro listed.

LED_OFF	Specifies LED is to be turned off.
LED_ON	Specifies LED is to be turned on.

LED Power Control Functions

The controller board can disable the LEDs on the controller board, the 5601, 5602 or 5604 I/O modules and the 5000 I/O modules to conserve power. This is particularly useful in solar powered or unattended installations. Refer to the ***SCADAPack System Hardware Manual*** for further information on LED power control.

There are four library functions related to LED power control. Refer to the ***Function Specification*** section for details on each function listed.

ledGetDefault	Get default LED power state
ledPower	Set LED power state
ledPowerSwitch	Read LED power switch
ledSetDefault	Set default LED power state

LED Power Control Macros

The **ctools.h** file defines the following macros for use with LED power control. Refer to the ***C Tools Macros*** section for details on each macro listed.

LED_OFF	Specifies LED is to be turned off.
LED_ON	Specifies LED is to be turned on.

LED Power Control Structure

The **ctools.h** file defines the structure **LED Power Control Structure** for LED power control information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Software Timer Functions

The controller provides 32 powerful software timers, which greatly simplify the task of programming time-related functions. Uses include:

- generation of time delays
- timing of process events such as tank fill times
- generation of time-based interrupts to schedule regular activities
- control of digital outputs by time periods

The 32 timers are individually programmable for tick rates from ten per second to once every 25.5 seconds. Time periods from 0.1 second to greater than nineteen days can be measured and controlled.

Timers operate in the background from a hardware interrupt generated by the main system clock. Once loaded, they count without intervention from the main program.

There are four library functions related to timers. Refer to the ***Function Specification*** section for details on each function listed.

interval	Set timer tick interval in tenths of seconds.
settimer	Set a timer. Timers count down from the set value to zero.
timer	Read the time period remaining in a timer.
read_timer_info	Read information about a software timer.

Software Timer Macros

The **ctools.h** file defines the following macros for use with timers. Refer to the ***C Tools Macros*** section for details on each macro listed.

NORMAL	Specifies normal count down timer.
TIMED_OUT	Specifies timer is has reached zero.
TIMER_BADINTERVAL	Error code indicating invalid timer interval.
TIMER_BADTIMER	Error code indicating invalid timer.
TIMER_BADVALUE	Error code indicating invalid time value.
TIMER_MAX	Number of last valid software timer.

Timer Information Structure

The **ctools.h** file defines the structure **Timer Information** for timer information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Timer Example Programs

Example 1: Turn on a digital output assigned to coil register 1 and wait 5 seconds before turning it off.

```
interval(0,10);     /* timer 0 tick rate = 1 second */
request_resource(IO_SYSTEM);
setdbase(MODBUS, 1, 1); /* turn on output */
release_resource(IO_SYSTEM);
```

```
settimer(0,5);      /* load timer 0 with 5 seconds */
while(timer(0))     /* wait until time expires */
{
	/* Allow other tasks to execute */
	release_processor();
}
request_resource(IO_SYSTEM);
setdbase(MODBUS, 1, 0);    /* shut off output */
release_resource(IO_SYSTEM);
```

Example 2: Time the duration a contact is on but wait in loop to measure time. Contact is assigned to status register 10001.

```
interval(0,1);      /* tick rate = 0.1 second */
request_resource(IO_SYSTEM);
if (dbase(MODBUS, 10001)) /* test if contact is on */
{
	settimer(0,63000);  /* start timer */
	while(dbase(MODBUS, 10001)) /* wait for turn off */
	{
		/* Allow other tasks to execute */
		release_resource(IO_SYSTEM);
		release_processor();
		request_resource(IO_SYSTEM);
	}
	printf("time period = %u\r\n",63000-timer(0));
}
release_resource(IO_SYSTEM);
```

Example 3: Open valve to fill tank and print alarm message if not full in 1 minute. Contact is assigned to status register 10001. Valve is controlled by coil register 1.

```
interval(0,10);     /* timer 0 tick rate = 1 second */
request_resource(IO_SYSTEM);
setdbase(MODBUS, 1, 1);          /* open valve */
settimer(0,60);     /* set timer for 1 minute */

/* tank not full if contact is off */
while((dbase(MODBUS, 10001)== 0) && timer(0))
{
	/* Allow other tasks to execute */
	release_resource(IO_SYSTEM);
	release_processor();
	request_resource(IO_SYSTEM);
}


if (dbase(MODBUS, 10001)== 0)
	puts("tank is not filling!!\r\n");
else
	puts("tank full\r\n");

setdbase(MODBUS, 1, 0); /* close valve */
release_resource(IO_SYSTEM);
```

Real Time Clock Functions

The controller is provided with a hardware based real time clock that independently maintains the time and date for the operating system. The time and date remain accurate during power-off. This allows the controller to be synchronized to time of day for such functions as shift production reports, automatic instrument calibration, energy logging, etc. The calendar can be used to automatically take the controller off-line during weekends and holidays. The calendar automatically handles leap years.

There are eight library functions, which access the real-time clock. Refer to the ***Function Specification*** section for details on each function listed.

alarmIn	Returns absolute time of alarm given elapsed time
getclock	Read the real time clock.
getClockAlarm	Reads the real time clock alarm settings.
getClockTime	Read the real time clock.
installClockHandler	Installs a handler for real time clock alarms.
resetClockAlarm	Resets the real time clock alarm so it will recur at the same time next day.
setclock	Set the real time clock.
setClockAlarm	Sets real time clock alarm.

Real Time Clock Macros

The **ctools.h** file defines the following macros for real time clock alarms. Refer to the ***C Tools Macros*** section for details on each macro listed.

AT_ABSOLUTE	Specifies a fixed time of day alarm.
AT_NONE	Disables alarms

Real Time Clock Structures

The **ctools.h** file defines the structures **Real Time Clock Structure** and **Alarm Settings Structure** for real time clock information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Real Time Clock Program Example

The following program illustrates how the date and time can be set and displayed. All fields of the clock structure need to be set with valid values for the clock to operate properly.

```
#include <ctools.h>

void main(void)
{
	struct clock now;
```

```
        /* Set to 12:01:00 on January 1, 1994 */

        now.hour      = 12;        /* set the time */
        now.minute    = 1;
        now.second    = 0;
        now.day       = 1;         /* set the date */
        now.month     = 1;
        now.year      = 94;
        now.dayofweek = 6;      /* day is Sat. */

        request_resource(IO_SYSTEM);
        setclock(&now);

        now = getclock();
        release_resource(IO_SYSTEM);

        /* Display current hour, minute and second */
        printf("%2d:%2d:%2d", now.hour, now.minute,
                     now.second);
}
```

The Jiffy Clock

The jiffy clock is a counter that increments 60 times per second. The jiffy clock is useful for measuring execution times or generating delays where a fine time base is required. The clock is reset to zero each time power is applied to the controller. It rolls over to zero after it reaches a value of 5183999. This is the number of 1/60-second intervals in 24 hours.

There are two library functions, which access the real-time clock. Refer to the ***Function Specification*** *C Function Library* chapter for a complete description.

setjiffy	set the jiffy clock
jiffy	read the jiffy clock

Watchdog Timer Functions

A watchdog timer is a hardware device, which enables rapid detection of computer hardware or software problems. In the event of a major problem, the CPU resets and the application program restarts.

The controller provides an integral watchdog timer for reliable operation. The watchdog timer resets the CPU if it detects a problem in either the hardware or system firmware. A user program can take control of the watchdog timer, so it will detect abnormal execution of the program.

A watchdog timer is a retriggerable, time delay timer. It begins a timing sequence every time it receives a reset pulse. The time delay is adjusted so that regular reset pulses prevent the timer from expiring. If the reset pulses cease, the watchdog timer expires and turns on its output, signifying a malfunction. The timer output in the controller resets the CPU and turns off all outputs at the I/O system.

The watchdog timer is normally reset by the operating system. This is transparent to the application program. Operating in such a fashion, the watchdog timer detects any hardware or firmware problems.

The program takes control of the timer, and resets it regularly. If unexpected operation of the program occurs, the reset pulses cease, and the watchdog timer resets the CPU. The program restarts from the beginning.

There are three library functions related to the watchdog timer. Refer to the ***Function Specification*** section for details on each function listed.

wd_auto	Gives control of the watchdog timer to the operating system (default).
wd_manual	Gives control of the watchdog timer to an application program.
wd_pulse	Generates a watchdog reset pulse.

A watchdog reset pulse must be generated at least every 500 ms. The CPU resets, and program execution starts from the beginning of the program, if the watchdog timer is not reset.

Watchdog Timer Program Example

The following program segment shows how the watchdog timer could be used to detect the unexpected operation of a section of a program.

```
wd_manual();  /* take control of watchdog timer */
do {
       /* program code */
       wd_pulse();         /* reset the watchdog timer */
}
while (condition)
wd_auto();                 /* return control to OS */
```

Always pass control of the watchdog timer back to the operating system before stopping a program, or switching to another task that expects the operating system to reset the timer.

Checksum Functions

To simplify the implementation of self-checking communication algorithms, the C Tools provide four types of checksums: additive, CRC-16, CRC-CCITT, and byte-wise exclusive-OR. The CRC algorithms are particularly robust, employing various polynomial methods to detect nearly all communication errors. Additional types of checksums are easily implemented using library functions.

There are two library functions related to checksums. Refer to the ***Function Specification*** section for details on each function listed.

checksum	Calculates additive, CRC-16, CRC-CCITT and exclusive-OR type checksums
crc_reverse	Calculates custom CRC type checksum using reverse CRC algorithm.

Checksum Macros

The **ctools.h** file defines macros for specifying checksum types. Refer to the ***C Tools Macros*** section for details on each macro listed.

ADDITIVE	Additive checksum
BYTE_EOR	Byte-wise exclusive OR checksum
CRC_16	CRC-16 type CRC checksum (reverse algorithm)
CRC_CCITT	CCITT type CRC checksum (reverse algorithm)

Serial Communication

The SCADAPack family of controllers offers three or four RS-232 serial ports. The Micro16 has two RS-232 serial communication ports. (com1 on controllers is also available as an RS-485 port.) The ports are configurable for baud rate, data bits, stop bits, parity and communication protocol.

To optimize performance, minimize the length of messages on com3 and com4. Examples of recommended uses for com3 and com4 are for local operator display terminals, and for programming and diagnostics using the IEC 61131-3 program.

For the SCADAPack 4000 programmable controllers, com1 is not available for C applications.

Default Serial Parameters

All ports are configured at reset with default parameters when the controller is powered up in SERVICE mode. The ports use stored parameters when the controller is reset in the RUN mode. The default parameters are listed below.

Parameter	com1	com2	Com3	Com4
Baud rate	9600	9600	9600	9600
Parity	none	none	None	None
Data bits	8	8	8	8
Stop bits	1	1	1	1
Duplex	full	full	Half	Half
Protocol	Modbus RTU	Modbus RTU	Modbus RTU	Modbus RTU
Station	1	1	1	1
Rx flow control	off	off	Rx disable	Rx disable
Tx flow control	off	off	Off	Off
Serial time out	60 s	60 s	60 s	60 s
Type	RS-232	RS-232	RS-232	RS-232

Serial Communication Time Out

When the controller is transmitting data on the communication ports, the transmit buffer may become full due to receipt of an XOFF character, a slow baud rate, or hardware handshaking.

If the transmit buffers become full, the task transmitting data is blocked until space is available or the serial time out period expires. If no space is available at the conclusion of this time period, the transmit buffer is emptied. The task then continues execution.

Debugging Serial Communication

Serial communication can be difficult to debug. This section describes the most common causes of communication failures.

- To communicate, the controller and an external device needs to use the same communication parameters. Check the parameters in both units.
- If some but not all characters transmit properly, you probably have a parity or stop bit mismatch between the devices.

The connection between two RS-232 Data Terminal Equipment (DTE) devices is made with a null-modem cable. This cable connects the transmit data output of one device to the receive data input of the other device – and vice versa. The controller is a DTE device. This cable is described in the ***System Hardware Manual*** for your controller.

The connection between a DTE device and a Data Communication Equipment (DCE) device is made with a straight cable. The transmit data output of the DTE device is connected to the transmit data input of the DCE device. The receive data input of the DTE device is connected to the receive data output of the DCE device. Modems are usually DCE devices. This cable is described in the ***System Hardware Manual*** for your controller.

Many RS-232 devices require specific signal levels on certain pins. Communication is not possible unless the required signals are present. In the controller the CTS line needs to be at the proper level. The controller will not transmit if CTS is OFF. If the CTS line is not connected, the controller will force it to the proper value. If an external device controls this line, it needs to turn it ON for the controller to transmit.

Serial Communication Functions

The **ctools.h** file defines the following serial communication related functions. Refer to the ***Function Specification*** section for details on each function listed. Additional serial communication functions are included in the Microtec run-time library.

clear_errors	Clear serial port error counters.
clear_tx	Clear serial port transmit buffer.
get_port	Read serial port communication parameters.

GetPortCharacteristics	Read information about features supported by a serial port.
get_status	Read serial port status and error counters.
install_handler	Install serial port character received handler.
portConfiguration	Get pointer to port configuration table
portIndex	Get array index for serial port
portStream	Get serial port corresponding to index
queue_mode	Set serial port transmitter mode.
route	Redirect standard I/O streams.
setDTR	Control RS232 port DTR signal.
set_port	Set serial port communication parameters.

Serial Communication Macros

The **ctools.h** file defines macros for specifying serial communication parameters. Refer to the ***C Tools Macros*** section for details on each macro listed.

BAUD75	Specifies 75-baud port speed.
BAUD110	Specifies 110-baud port speed.
BAUD150	Specifies 150-baud port speed.
BAUD300	Specifies 300-baud port speed.
BAUD600	Specifies 600-baud port speed.
BAUD1200	Specifies 1200-baud port speed.
BAUD2400	Specifies 2400-baud port speed.
BAUD4800	Specifies 4800-baud port speed.
BAUD9600	Specifies 9600-baud port speed.
BAUD19200	Specifies 19200-baud port speed.
BAUD38400	Specifies 38400-baud port speed.
BAUD57600	Specifies 57600-baud port speed.
BAUD115200	Specifies 115200-baud port speed.
com1	Points to a file object for *com1* serial port.
com2	Points to a file object for *com2* serial port.
com3	Points to a file object for *com3* serial port.
com4	Points to a file object for *com4* serial port.
DATA7	Specifies 7 bit world length.
DATA8	Specifies 8 bit word length.

DISABLE	Specifies flow control is disabled.
ENABLE	Specifies flow control is enabled.
EVEN	Specifies even parity.
FULL	Specifies full duplex.
FOPEN_MAX	Redefinition of macro from stdio.h
HALF	Specifies half duplex.
NONE	Specifies no parity.
NOTYPE	Specifies serial port type is not known.
ODD	Specifies odd parity.
PC_FLOW_RX_RECEIVE_STOP	Receiver disabled after receipt of a message.
PC_FLOW_RX_XON_XOFF	Receiver Xon/Xoff flow control.
PC_FLOW_TX_IGNORE_CTS	Transmitter flow control ignores CTS.
PC_FLOW_TX_XON_XOFF	Transmitter Xon/Xoff flow control.
RS232	Specifies serial port is an RS-232 port.
RS232_MODEM	Specifies serial port is an RS-232 dial-up modem.
RS485_4WIRE	Specifies serial port is a 4 wire RS-485 port.
SERIAL_PORTS	Number of serial ports.
SIGNAL_CTS	I/O line bit mask: clear to send signal
SIGNAL_DCD	I/O line bit mask: carrier detect signal
SIGNAL_OFF	Specifies a signal is de-asserted
SIGNAL_OH	I/O line bit mask: off hook signal
SIGNAL_ON	Specifies a signal is asserted
SIGNAL_RING	I/O line bit mask: ring signal
SIGNAL_VOICE	I/O line bit mask: voice/data switch signal
STOP1	Specifies 1 stop bit.
STOP2	Specifies 2 stop bits.

Serial Communication Structures

The **ctools.h** file defines the structures **Serial Port Configuration**, **Serial Port Status** and **Serial Port Characteristics** for serial port configuration and information. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Microtec Serial I/O Functions

These library functions are related to serial communication. They are documented in the *Microtec MCCM77 Documentation Set*.

fgetc	reads a character from a stream
fgets	reads a string from a stream
fputc	writes a character to a stream
fputs	writes a string to a stream
fread	reads from a stream
fwrite	writes to a stream
getc	reads a character from a stream
getchar	reads a character from standard input device
gets	reads a string from a stream
initport	re-initializes serial port
printf	formatted output to a stream
putc	writes a character to a stream
putchar	reads a character to standard output device
puts	writes a string to a stream
scanf	formatted input from a stream

Dial-Up Modem Functions

These library functions provide control of dial-up modems. They are used with external modems connected to a serial port. An external modem normally connects to the RS-232 port with a DTE to DCE cable. Consult the ***System Hardware Manual*** for your controller for details. Refer to the ***Function Specification*** section for details on each function listed.

modemInit	send initialization string to dial-up modem.
modemInitStatus	read status of modem initialization operation.
modemInitEnd	terminate modem initialization operation.
modemDial	connect with an external device using a dial-up modem.
modemDialStatus	read status of connection with external device using a dial-up modem.
modemDialEnd	terminate connection with external device using a dial-up modem.
modemAbort	unconditionally terminate connection with external device or modem initialization (used in task exit handler).

modemAbortAll unconditionally terminate connections with external device or modem initializations (used in task exit handler).

modemNotification notify the dial-up modem handler that an interesting event has occurred. This function is usually called whenever a message is received by a protocol.

Dial-Up Modem Macros

The **ctools.h** file defines the following macros of interest to a C application program. Refer to the ***C Tools Macros*** section for details on each macro listed.

MODEM_CMD_MAX_LEN Maximum length of the modem initialization command string

PHONE_NUM_MAX_LEN Maximum length of the phone number string

Dial-Up Modem Enumeration Types

The ctools.h file defines the enumerated types DialError and DialState. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Dial-up Modem Structures

The ctools.h file defines the structures ModemInit and ModemSetup**.** Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Modem Initialization Example

The following code shows how to initialize a modem. Typically, the modem initialization is used to prepare a modem to answer calls. The example sets up a Hayes modem to answer incoming calls.

```
#include <ctools.h>

void main(void)
{
struct ModemInit initSettings;
reserve_id portID;
enum DialError status;
enum DialState state;
struct pconfig portSettings;

/* Configure serial port 1 */
portSettings.baud      = BAUD1200;
portSettings.duplex    = FULL;
portSettings.parity    = NONE;
portSettings.data_bits = DATA8;
portSettings.stop_bits = STOP1;
portSettings.flow_rx   = DISABLE;
portSettings.flow_tx   = DISABLE;
portSettings.type      = RS232_MODEM;
portSettings.timeout   = 600;
```

```
request_resource(IO_SYSTEM);
set_port(com1, &portSettings);
release_resource(IO_SYSTEM);

/* Initialize Hayes modem to answer incoming calls */
initSettings.port = com1;
strcpy(initSettings.modemCommand, " F1Q0V1X1 S0=1");
if (modemInit(&initSettings, &portID) == DE_NoError)
{
      do
      {
            /* Allow other tasks to execute */
            release_processor();

            /* Wait for the initialization to complete */
            modemInitStatus(com1, portID, &status, &state);
      }
      while (state == DS_Calling);

      /* Terminate the initialization */
      modemInitEnd(com1, portID, &status);
}
}
```

Connecting with a Remote Controller Example

The following code shows how to connect to a remote controller using a modem. The example uses a US Robotics modem. It also demonstrates the use of the modemAbort function in an exit handler.

```
#include <ctools.h>

/* -----------------------------------------------
   The shutdown function aborts any active
   modem connections when the task is ended.
   ----------------------------------------------- */
void shutdown(void)
{
      modemAbort(com1);
}

void main(void)
{
struct ModemSetup dialSettings;
reserve_id portID;
enum DialError status;
enum DialState state;
struct pconfig portSettings;
TASKINFO taskStatus;

/* Configure serial port 1 */
portSettings.baud      = BAUD19200;
portSettings.duplex    = FULL;
portSettings.parity    = NONE;
portSettings.data_bits = DATA8;
portSettings.stop_bits = STOP1;
```

```
portSettings.flow_rx   = DISABLE;
portSettings.flow_tx   = DISABLE;
portSettings.type      = RS232_MODEM;
portSettings.timeout   = 600;
request_resource(IO_SYSTEM);
set_port(com1, &portSettings);
release_resource(IO_SYSTEM);

/* Configure US Robotics modem */
dialSettings.port         = com1;
dialSettings.dialAttempts = 3;
dialSettings.detectTime   = 60;
dialSettings.pauseTime    = 30;
dialSettings.dialmethod   = 0;
strcpy(dialSettings.modemCommand, "&F1 &A0 &K0 &M0 &B1");
strcpy(dialSettings.phoneNumber, "555-1212");

/* set up exit handler for this task */
taskStatus = getTaskInfo(0);
installExitHandler(taskStatus.taskID, shutdown);

/* Connect to the remote controller */
if (modemDial(&dialSettings, &portID) == DE_NoError)
{
       do
       {
              /* Allow other tasks to execute */
              release_processor();

              /* Wait for initialization to complete */
              modemDialStatus(com1, portID, &status, &state);
       }
       while (state == DS_Calling);

       /* If the remote controller connected */
       if (state == DS_Connected)
       {
              /* Talk to remote controller here */
       }

       /* Terminate the connection */
       modemDialEnd(com1, portID, &status);
}
}
```

A pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

Communication Protocols

The TeleBUS protocols are compatible with the widely used Modbus RTU and ASCII protocols. The TeleBUS communication protocols provide a standard communication interface to SCADAPack controllers. Additional TeleBUS commands provide remote programming and diagnostics capability.

The TeleBUS protocols provide access to the I/O database in the controller. The I/O database contains user-assigned registers and general purpose registers. Assigned registers map directly to the I/O hardware or system parameter in the controller. General purpose registers can be used by ladder logic and C application programs to store processed information, and to receive information from a remote device.

The TeleBUS protocols operate on a wide variety of serial data links. These include RS-232 serial ports, RS-485 serial ports, radios, leased line modems, and dial up modems. The protocols are generally independent of the communication parameters of the link, with a few exceptions.

Application programs can initiate communication with remote devices. A multiple port controller can be a data concentrator for remote devices, by polling remote devices on one port(s) and responding as a slave on another port(s).

The protocol type, communication parameters and station address are configured separately for each serial port on a controller. One controller can appear as different stations on different communication networks. The port configuration can be set from an application program, from the Telepace programming software, or from another Modbus or DF1 compatible device.

Protocol Type

The protocol type may be set to emulate the Modbus ASCII and Modbus RTU protocols, or it may be disabled. When the protocol is disabled, the port functions as a normal serial port.

The DF1 option enables the emulation of the DF1 protocols.

The DNP (Distributed Network Protocol) option enables DNP. See the ***DNP Communication Protocol*** section for details on this protocol.

Station Number

The TeleBUS protocol allows up to 254 devices on a network using standard addressing and up to 65534 devices using extended addressing. Station numbers identify each device. A device responds to commands addressed to it, or to commands broadcast to all stations.

The station number is in the range 1 to 254 for standard addressing and 1 to 65534 for extended addressing. Address 0 indicates a command broadcast to all stations, and cannot be used as a station number. Each serial port may have a unique station number.

The TeleBUS DF1 protocols allow up to 255 devices on a network. Station numbers identify each device. A device responds to commands addressed to it, or to commands broadcast to all stations. The station number is in the range 0 to 254. Address 255 indicates a command broadcast to all stations, and cannot be used as a station number. Each serial port may have a unique station number.

Store and Forward Messaging

Store and forward messaging re-transmits messages received by a controller. Messages may be re-transmitted on any serial port, with or without station address translation. A user-defined translation table determines actions performed for each message. Store and forward messaging may be enabled or disabled on each port. It is disabled by default.

Store and forward messaging is not supported by DNP or TeleBUS DF1 protocol.

Communication Protocols Functions

There are several library functions related to TeleBUS communication protocol. Refer to the ***Function Specification*** section for details on each function listed.

checkSFTranslationTable Check translation table for invalid entries.

clear_protocol_status Clears protocol message and error counters.

clearSFTranslationTable Clear all store and forward translation table entries.

enronInstallCommandHandler Installs handler for Enron Modbus commands.

getABConfiguration Reads DF1 protocol configuration parameters.

get_protocol Reads protocol parameters.

getProtocolSettings Reads extended addressing protocol parameters for a serial port.

getProtocolSettingsEx Reads extended addressing and Enron Modbus protocol parameters for a serial port.

get_protocol_status Reads protocol message and error counters.

getSFMapping This function is a stub and no longer performs a necessary operation.

getSFTranslation Read store and forward translation table entry.

installModbusHandler This function allows user-defined extensions to standard Modbus protocol.

master_message Sends a protocol message to another device.

modbusExceptionStatus Sets response for the read exception status function.

modbusSlaveID Sets response for the read slave ID function.

pollABSlave Requests a response from a slave controller using the half-duplex version of the protocol.

resetAllABSlaves Clears responses from the response buffers of half-duplex slave controllers.

setABConfiguration Defines DF1 protocol configuration parameters.

set_protocol Sets protocol parameters and starts protocol.

setProtocolSettings Sets extended addressing protocol parameters for a serial port.

setProtcolSettingEx Sets extended addressing and Enron Modbus protocol parameters for a serial port.

setSFMapping This function is a stub and no longer performs a necessary operation.

setSFTranslation Write store and forward translation table entry.

start_protocol Starts protocol execution based on stored parameters.

Communication Protocols Macros

The **ctools.h** file defines macros for specifying communication protocol parameters. Refer to the ***C Tools Macros*** section for details on each macro listed.

AB_FULL_BCC Specifies the DF1 Full Duplex protocol emulation for the serial port. (BCC checksum)

AB_FULL_CRC Specifies the DF1 Full Duplex protocol emulation for the serial port. (CRC checksum)

AB_HALF_BCC Specifies the DF1 Half Duplex protocol emulation for the serial port. (BCC checksum)

AB_HALF_CRC Specifies the DF1 Half Duplex protocol emulation for the serial port. (CRC checksum)

FORCE_MULTIPLE_COILS Modbus function code

FORCE_SINGLE_COIL Modbus function code

LOAD_MULTIPLE_REGISTERS Modbus function code

LOAD_SINGLE_REGISTER Modbus function code

MM_BAD_ADDRESS Master message status: invalid database address

MM_BAD_FUNCTION Master message status: invalid function code

MM_BAD_LENGTH Master message status: invalid message length

MM_BAD_SLAVE Master message status: invalid slave station address

MM_NO_MESSAGE Master message status: no message was sent.

MM_PROTOCOL_NOT_SUPPORTED Master message status: selected protocol is not supported.

MM_RECEIVED Master message status: response was received.

MM_SENT Master message status: message was sent.

MM_EOT Master message status: DF1 slave response was an EOT message

MM_WRONG_RSP Master message status: DF1 slave response did not match command sent

MM_CMD_ACKED Master message status: DF1 half duplex command has been acknowledged by slave – Master may now send poll command

MM_EXCEPTION_FUNCTION Master message status: Modbus slave returned a function exception

MM_EXCEPTION_ADDRESS Master message status: Modbus slave returned an address exception

MM_EXCEPTION_VALUE Master message status: Modbus slave returned a value exception

MM_RECEIVED_BAD_LENGTH Master message status: response received with incorrect amount of data.

MODBUS_ASCII Specifies the Modbus ASCII protocol emulation for the serial port.

MODBUS_RTU Specifies the Modbus RTU protocol emulation for the serial port.

NO_PROTOCOL Specifies no communication protocol for the serial port.

READ_COIL_STATUS Modbus function code

READ_EXCEPTION_STATUS Modbus function code

READ_HOLDING_REGISTER Modbus function code

READ_INPUT_REGISTER Modbus function code

READ_INPUT_STATUS Modbus function code

REPORT_SLAVE_ID Modbus function code

SF_ALREADY_DEFINED Result code: translation is already defined in the table

SF_INDEX_OUT_OF_RANGE Result code: invalid translation table index

SF_NO_TRANSLATION Result code: entry does not define a translation

SF_PORT_OUT_OF_RANGE Result code: serial port is not valid

SF_STATION_OUT_OF_RANGE Result code: station number is not valid

SF_TABLE_SIZE Number of entries in the store and forward table

SF_VALID Result code: translation is valid

Communication Protocols Enumeration Types

The **ctools.h** file defines the enumeration type **ADDRESS_MODE**. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Communication Protocols Structures

The **ctools.h** file defines the structures **Protocol Status Information**, **Protocol Settings**, **Extended Protocol Settings**, **Store and Forward Message** and **Store and Forward Status**. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

DNP Communication Protocol

DNP, the Distributed Network Protocol, is a standards-based communications protocol developed to achieve interoperability among systems in the electric utility, oil & gas and water/waste water industries. This robust, flexible non-proprietary protocol is based on existing open standards to work within a variety of networks. The IEEE has recommended DNP for remote terminal unit to intelligent electronic device messaging. DNP can also be implemented in any SCADA system for efficient and robust communications between substation computers, RTUs, IEDs and master stations; over serial or LAN-based systems.

DNP offers flexibility and functionality that go far beyond conventional communications protocols. Among its robust and flexible features DNP 3.0 includes:

- Output options
- Addressing for over 65,000 devices on a single link
- Time synchronization and time-stamped events
- Broadcast messages
- Data link and application layer confirmation

DNP 3.0 was originally designed based on three layers of the OSI seven-layer model: application layer, data link layer and physical layer. The application layer is object-based with objects provided for generic data formats. The data link layer provides for several methods of retrieving data such as polling for classes and object variations. The physical layer commonly defines a simple RS-232 or RS-485 interface.

Refer to the **DNP User Manual** for complete information on DNP protocol, including the **Device Profile Document**.

DNP Communication Protocols Functions

There are several library functions related to DNP communication protocol. Refer to the ***Function Specification*** section for details on each function listed.

dnpInstallConnectionHandler Configures the connection handler for DNP.

dnpClearEventLog Deletes all change events from the DNP change event buffers.

dnpConnectionEvent Report a DNP connection event

dnpCreateRoutingTable Allocates memory for a new routing table.

dnpGenerateEventLog Generates a change event for the DNP point.

dnpGetConfiguration Reads the DNP protocol configuration.

dnpGetConfigurationEx Reads the extended DNP configuration parameters.

dnpSaveConfiguration Writes the DNP protocol configuration parameters.

dnpSaveConfigurationEx Writes the extended DNP configuration parameters

dnpGetBIConfig Reads the configuration of a DNP binary input point.

dnpSaveBIConfig Writes the configuration of a DNP binary input point.

dnpSaveBIConfigEx Writes the configuration of an extended DNP Binary Input point

dnpGetBOConfig Reads the configuration of a DNP binary output point.

dnpGetBIConfigEx Reads the configuration of an extended DNP Binary Input point.

dnpSaveBOConfig Sets the configuration of a DNP binary output point.

dnpGetAI16Config Reads the configuration of a DNP 16-bit analog input point.

dnpSaveAI16Config Sets the configuration of a DNP 16-bit analog input point.

dnpGetAI32Config Reads the configuration of a DNP 32-bit analog input point.

dnpSaveAISFConfig Sets the configuration of a DNP 32-bit short floating analog input point

dnpGetAISFConfig Reads the configuration of a DNP 32-bit short floating analog input point.

dnpSaveAI32Config Sets the configuration of a DNP 32-bit analog input point.

dnpGetAO16Config Reads the configuration of a DNP 16-bit analog output point.

dnpSaveAO16Config Sets the configuration of a DNP 32-bit analog output point.

dnpGetAO32Config Reads the configuration of a DNP 32-bit analog output point.

dnpSaveAO32Config Sets the configuration of a DNP 32-bit analog output point.

dnpSaveAOSFConfig Sets the configuration of a DNP 32-bit short floating analog output point.

dnpGetAOSFConfig Sets the configuration of a DNP 32-bit short floating analog output point.

dnpGetCI16Config Reads the configuration of a DNP 16-bit counter input point.

dnpSaveCI16Config Sets the configuration of a DNP 16-bit counter input point.

dnpGetCI32Config Reads the configuration of a DNP 32-bit counter input point.

dnpSaveCI32Config Sets the configuration of a DNP 32-bit counter input point.

dnpGetRuntimeStatus Reads the current status of all DNP change event buffers.

dnpSendUnsolicited Sends an ‘Unsolicited Response’ message in DNP protocol.

dnpSendUnsolicitedResponse Sends an Unsolicited Response message in DNP, with data from the specified classes.

dnpWriteRoutingTableEntry Wwrites an entry in the DNP routing table.

dnpReadRoutingTableEntry Reads an entry from the routing table.

dnpReadRoutingTableSize Reads the total number of entries in the routing table.

dnpSearchRoutingTable Searches the routing table for a specific DNP address.

dnpWriteRoutingTableDialStrings Writes a primary and secondary dial string into an entry in the DNP routing table.

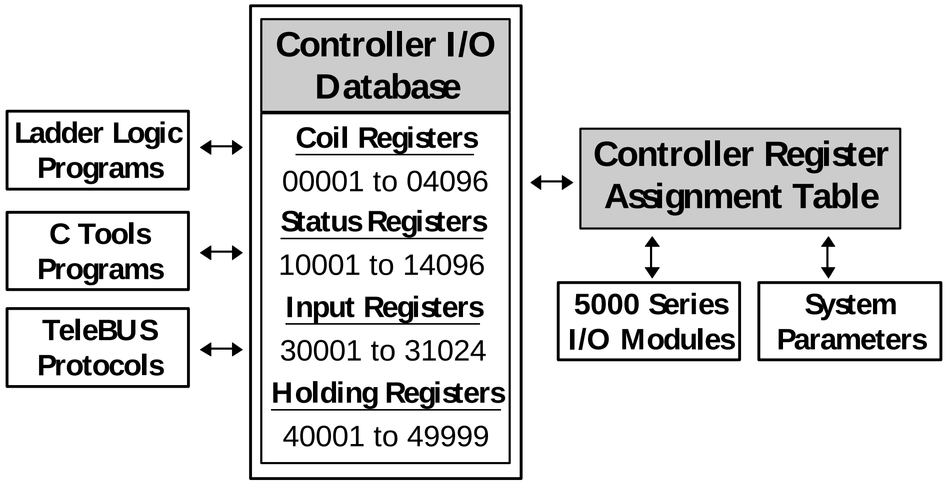
dnpReadRoutingTableDialStrings Reads a primary and secondary dial string from an entry in the DNP routing table.

DNP Communication Protocol Structures and Types

The **ctools.h** file defines the structures **DNP Configuration**, **Binary Input Point**, **Binary Output Point**, **Analog Input Point**, **Analog Output Point** and **Counter Input Point**. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

I/O Database

The I/O database allows data to be shared between C programs, Ladder Logic programs and communication protocols. A simplified diagram of the I/O Database is shown below.



The I/O database contains general purpose and user-assigned registers. General purpose registers may be used by Ladder Logic and C application programs to store processed information and to receive information from a remote device. Initially all registers in the I/O Database are general purpose registers.

User-assigned registers are mapped directly from the I/O database to physical I/O hardware, or to controller system configuration and diagnostic parameters. The Register Assignment performs the mapping of registers from the I/O database to physical I/O hardware and system parameters.

User-assigned registers are initialized to the default hardware state or system parameter when the controller is reset. Assigned output register values are not maintained during power failures. Assigned output registers do retain their values during application program loading.

General purpose registers retain their values during power failures and application program loading. The values change only when written by an application program or a communication protocol.

The TeleBUS communication protocols provide a standard communication interface to the controller. The TeleBUS protocols are compatible with the widely used Modbus RTU and ASCII protocols and provide access to the I/O database in the controller.

I/O Database Register Types

The I/O database is divided into four types of I/O registers. Each of these types are initially configured as general purpose registers by the controller.

Coil Registers

Coil registers are single bit registers located in the digital output section of the I/O database. Coil, or digital output, database registers may be assigned to 5000 I/O digital output modules or SCADAPack I/O modules through the Register Assignment. Coil registers may also be assigned to controller on-board digital outputs and to system configuration modules.

There are 4096 coil registers numbered 00001 to 04096. Ladder logic programs, C language programs, and the TeleBUS protocols can read from and write to these registers.

Status Registers

Status registers are single bit registers located in the digital input section of the I/O database. Status, or digital input, database registers may be assigned to 5000 I/O digital input modules or SCADAPack I/O modules through the Register Assignment. Status registers may also be assigned to controller on-board digital inputs and to system diagnostic modules.

There are 4096 status registers are numbered 10001 to 14096. Ladder logic programs and the TeleBUS protocols can only read from these registers. C language application programs can read data from and write data to these registers.

Input Registers

Input registers are 16 bit registers located in the analog input section of the I/O database. Input, or analog input, database registers may be assigned to 5000 I/O analog input modules or SCADAPack I/O modules through the Register Assignment. Input registers may also be assigned to controller internal analog inputs and to system diagnostic modules.

There are 1024 input registers numbered 30001 to 31024. Ladder logic programs and the TeleBUS protocols can only read from these registers. C language application programs can read data from and write data to these registers.

The I/O database for the SCADAPack 100 controller has 512 input registers numbered 30001 to 30512. Ladder logic programs and the TeleBUS protocols can only read from these registers. C language programs can read data from and write data to these registers.

Holding Registers

Holding registers are 16 bit registers located in the analog output section of the I/O database. Holding, or analog output, database registers may be assigned to 5000 I/O analog output modules or SCADAPack analog output modules through the Register Assignment. Holding registers may also be assigned to system diagnostic and configuration modules.

There are 9999 input registers numbered 40001 to 49999. Ladder logic programs, C language programs, and the TeleBUS protocols can read from and write to these registers.

The I/O database for the SCADAPack 100 controller has 4000 holding registers numbered 40001 to 44000. Ladder logic programs, C language programs, and the TeleBUS protocols can read from and write to these registers.

I/O Database Functions

There are two library functions related to the I/O database. Refer to the ***Function Specification*** section for details on each function listed.

dbase	Reads a value from the I/O database.
setdbase	Writes a value to the I/O database.

I/O Database Macros

The **ctools.h** file defines library functions for the I/O database. Refer to the ***C Tools Macros*** section for details on each macro listed.

AB	Specifies Allan-Bradley database addressing.
DB_BADSIZE	Error code: out of range address specified
DB_BADTYPE	Error code: bad database addressing type specified
DB_OK	Error code: no error occurred
LINEAR	Specifies linear database addressing.
MODBUS	Specifies Modbus database addressing.
NUMAB	Number of registers in the Allan-Bradley database.
NUMCOIL	Number of registers in the Modbus coil section.
NUMHOLDING	Number of registers in the Modbus holding register section.
NUMINPUT	Number of registers in the Modbus input registers section.
NUMLINEAR	Number of registers in the linear database.
NUMSTATUS	Number of registers in the Modbus status section.
START_COIL	Start of the coil section in the linear database.
START_HOLDING	Start of the holding registers section in the linear database.
START_INPUT	Start of the input register section in the linear database.
START_STATUS	Start of the status section in the linear database.

Register Assignment Functions

I/O hardware that is used by the controller need to be assigned to I/O database registers in order for these I/O points to be scanned continuously. I/O data may then be accessed through the I/O database within the C program. C programs may read data from, or write data to the I/O hardware through user- assigned registers in the I/O database.

The Register Assignment assigns I/O database registers to user-assigned registers using I/O modules. An I/O Module can refer to an actual I/O hardware module (e.g. *5401 Digital Input Module*) or it may refer to a set of controller parameters, such as serial port settings.

The chapter *Register Assignment Reference* of the ***Telepace Ladder Logic Reference and User Manual*** contains a description of what each module is used for and the register assignment requirements for the I/O module.

Register assignments configured using the Telepace *Register Assignment* dialog may be stored in the Telepace program file or downloaded directly to the controller. To obtain error checking that avoids invalid register assignments, use the *Telepace Register Assignment* dialog to initially build the Register Assignment. The Register Assignment can then be saved in a Ladder Logic file (e.g. filename.lad) and downloaded with the C program.

There are several library functions related to register assignment. Refer to the ***Function Specification*** section for details on each function listed.

clearRegAssignment Erases the current Register Assignment.

addRegAssignment Adds one I/O module to the current Register Assignment.

getIOErrorIndication Gets the control flag for the I/O module error indication

getOutputsInStopMode Gets the control flags for state of Outputs in Ladders Stop Mode

setIOErrorIndication Sets the control flag for the I/O module error indication

setOutputsInStopMode Sets the control flags for state of Outputs in Ladders Stop Mode

Register Assignment Enumeration Types

The **ctools.h** file defines one enumeration type. The **ioModules** enumeration type defines a list of results of sending a command. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

Register Assignment Structure

The **ctools.h** file defines the structure **RegAssign.** Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

HART Communication

The HART ® protocol is a field bus protocol for communication with smart transmitters.

The HART protocol driver provides communication between Micro16 and SCADAPack controllers and HART devices. The protocol driver uses the model 5904 HART modem for communication. Four HART modem modules are supported per controller.

The driver allows HART transmitters to be used with C application programs and with IEC 61131-3. The driver can read data from HART devices.

HART Command Functions

The **ctools.h** file defines the following HART command related functions. Refer to the ***Function Specification*** section for details on each function listed.

hartIO	Reads data from the 5904 interface module, processes HART responses, processes HART commands, and writes commands and configuration data to the 5904 interface module.
hartCommand	send a HART command string and specify a function to handle the response
hartCommand0	read unique identifier using short-address algorithm
hartCommand1	read primary variable
hartCommand2	read primary variable current and percent of span
hartCommand3	read primary variable current and dynamic variables
hartCommand11	read unique identifier associated with tag
hartCommand33	read specified transmitter variables
hartStatus	return status of last HART command sent
hartGetConfiguration	read HART module settings
hartSetConfiguration	write HART module settings
hartPackString	convert string to HART packed string
hartUnpackString	convert HART packed string to string

HART Command Macros

The **ctools.h** file defines the following macro of interest to a C application program. Refer to the ***C Tools Macros*** section for details.

DATA_SIZE	Maximum length of the HART command or response field.

HART Command Enumeration Types

The **ctools.h** file defines one enumeration type. The **HART_RESULT** enumeration type defines a list of results of sending a command. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

HART Command Structures

The **ctools.h** file defines five structures. Refer to the ***C Tools Structures and Types*** section for complete information on structures and enumeration types.

The HART_DEVICE type is a structure containing information about the HART device.

The HART_VARIABLE type is a structure containing a variable read from a HART device.

The HART_SETTINGS type is a structure containing the configuration for the HART modem module.

The HART_COMMAND type is a structure containing a command to be sent to a HART slave device.

The HART_RESPONSE type is a structure containing a response from a HART slave device.

PID Control

Telepace C Tools provides a total of 32 independent PID (Proportional, Integral, and Derivative) controllers. PID control blocks operate independent of application programs. An elaborate control program need not be written to use the control blocks. A simple program to set up the control blocks is all that is required.

The PID control blocks are not limited to the PID control algorithm. They also provide ratio control, ratio/bias control, alarm scanning and square root functions. Control blocks may be interconnected to exchange setpoints, output limits, and other parameters.

Refer to the PID Controllers section of the Telepace Ladder Logic User Manual for complete information on configuring and using PID controllers.

PID Control Functions

The **ctools.h** file defines the following PID control related functions. Refer to the ***Function Specification*** section for details on each function listed.

auto_pid	Set a PID block to execute automatically at the specified rate.
clear_pid	Set all PID block variables to zero.
get_pid	This function returns the value of a PID control block variable.
set_pid	This function assigns *value* to a PID control block variable.

PID Control Macros

The **ctools.h** file defines the following macros for PID block access. Refer to the ***C Tools Macros*** section for details on each function listed.

AO	Variable name: alarm output address
CA	Variable name: cascade setpoint source
CR	Variable name: control register
DB	Variable name: deadband
DO	Variable name: decrease output
ER	Variable name: error
EX	Variable name: automatic execution period
FS	Variable name: full scale output limit
GA	Variable name: gain

HI	Variable name: high alarm setpoint
IB	Variable name: input bias
IH	Variable name: inhibit execution address
IN	Variable name: integrated error
IO	Variable name: increase output
IP	Variable name: input source
LO	Variable name: low alarm setpoint
OB	Variable name: output bias
OP	Variable name: output
PE	Variable name: period
PID_ALARM	Control register mask: alarms enabled
PID_ALARM_ABS	Control register mask: absolute alarms
PID_ALARM_ACK	Status register mask: alarm acknowledged
PID_ALARM_DEV	Control register mask: deviation alarms
PID_ALARM_ONLY	Control register mask: alarm only block
PID_ALARM_RATE	Control register mask: rate alarms
PID_ANALOG_IP	Control register mask: analog input
PID_ANALOG_OP	Control register mask: analog output
PID_BAD_BLOCK	Return code: bad block number specified.
PID_BAD_IO_IP	Status register mask: I/O failure on block input
PID_BAD_IO_OP	Status register mask: I/O failure on block output
PID_BLOCK_IP	Control register mask: input from output of another block
PID_BLOCKS	Number of PID blocks.
PID_CLAMP_FULL	Status register mask: output is clamped at full scale
PID_CLAMP_ZERO	Status register mask: output is clamped at zero scale
PID_ER_SQR	Control register mask: take square root of error
PID_HI_ALARM	Status register mask: high alarm detected
PID_INHIBIT	Status register mask: external inhibit input is on
PID_LO_ALARM	Status register mask: low alarm detected
PID_MANUAL	Status register mask: block is in manual mode
PID_MODE_AUTO	Control register mask: automatic mode
PID_MODE_MANUAL	Control register mask: manual mode
PID_MOTOR_OP	Control register mask: motor pulse duration output

PID_NO_ALARM	Control register mask: alarms disabled
PID_NO_ER_SQR	Control register mask: normal error
PID_NO_IP	Control register mask: no input (other than IP)
PID_NO_OP	Control register mask: no output
PID_NO_PV_SQR	Control register mask: normal PV
PID_NO_SP_TRACK	Control register mask: setpoint tracking disabled
PID_OK	Return code: operation completed successfully.
PID_OUT_DB	Status register mask: PID controller outside of deadband
PID_PID	Control register mask: PID control block
PID_PULSE_OP	Control register mask: pulse duration output
PID_PV_SQR	Control register mask: take square root of PV
PID_RATE_CLAMP	Status register mask: rate gain clamed at maximum
PID_RATIO_BIAS	Control register mask: ratio/bias control block
PID_RUNNING	Status register mask: block is executing
PID_SP_CASCADE	Control register mask: cascade setpoint
PID_SP_NORMAL	Control register mask: setpoint stored in SP
PID_SP_TRACK	Control register mask: setpoint tracking enabled
PV	Variable name: process value
RA	Variable name: rate time
RE	Variable name: reset time
SP	Variable name: setpoint
SR	Variable name: status register
ZE	Variable name: zero scale output limit

Backward Compatibility Functions

The following functions are provided for backward compatibility. They cannot access 5000 I/O modules. It is recommended that these functions not be used in new programs. Instead use Register Assignment or call the specific I/O module driver function directly.

These functions are defined in **ctools.h** for backward compatibility with these programs.

ain	Reads analog input
aioError	Reads analog I/O communication status
aout	Writes analog output
counter	Reads counter module input channel

counterError	Reads counter module error flag
din	Reads digital input channel (8 I/O points)
dout	Writes digital output channel (8 I/O points)
off	Tests If one digital I/O point is OFF
on	Tests If one digital I/O point is ON
pulse	Generates a square wave on a digital output point
pulse_train	Generates a series of pulses on a digital output point
timeout	Performs time delayed action on a digital output point
turnoff	Writes one digital output point to OFF status
turnon	Writes one digital output point to ON status

Backward Compatibility Macros

The following macros may have been used in C programs written for a controller with firmware version 1.22 or older to support the functions: ain, aioError, aout, counter, counterError, din, dout, off, on, pulse, pulse_train, timeout, turnoff or turnon.

These macros are defined in **ctools.h** for backward compatibility with these programs.

AIN_END	Number of last analog input channel.
AIN_START	Number of first analog input channel.
AIO_BADCHAN	Error code: bad analog input channel specified.
AIO_TIMEOUT	Error code: input device did not respond.
AIO_SUPPORTED	If defined indicates analog I/O supported.
AOUT_END	Number of last analog output channel.
AOUT_START	Number of first analog output channel.
COUNTER_CHANNELS	Specifies number of 5000 I/O counter input channels
COUNTER_END	Number of last counter input channel
COUNTER_START	Number of first counter input channel
COUNTER_SUPPORTED	If defined indicates counter I/O hardware supported.
DIN_END	Number of last regular digital input channel.
DIN_START	Number of first regular digital input channel
DIO_SUPPORTED	If defined indicates digital I/O hardware supported.
DOUT_END	Number of last regular digital output channel.
DOUT_START	Number of first regular digital output channel

DUTY_CYCLE	Specifies timer is generating square wave output.
EXTENDED_DIN_END	Number of last extended digital input channel.
EXTENDED_DIN_START	Number of first extended digital input channel
EXTENDED_DOUT_END	Number of last extended digital output channel.
EXTENDED_DOUT_START	Number of first extended digital output channel
NORMAL	Specifies normal count down timer.
PULSE_TRAIN	Specifies timer is generating pulse train output.
TIMEOUT	Specifies timer is generating timed output change.
TIMER_BADADDR	Error code: invalid digital I/O address.

Telepace C Tools Function Specifications

The controller C function specifications are formatted as follows. The functions are listed alphabetically.

Name	Each specification begins with the name of the function and a brief description.
Syntax	The syntax shows a prototype for the function, indicating the return type and the types of its arguments. Any necessary header files are listed.
Description	This defines the calling parameters for the function and its return values.
Notes	This section contains additional information on the function, and considerations for its use.
See Also	This section lists related functions.
Example	The example gives a brief sample of the use of the function.

addRegAssignment

Add Register Assignment

Syntax

```
#include <ctools.h>
unsigned addRegAssignment(
       unsigned moduleType,
       unsigned moduleAddress,
       unsigned startingRegister1,
       unsigned startingRegister2,
       unsigned startingRegister3,
       unsigned startingRegister4);
```

Description

The **addRegAssignment** function adds one I/O module to the current Register Assignment of type *moduleType*. The following symbolic constants are valid values for *moduleType*:

AIN_520xTemperature	DIAG_forceLED
AIN_520xRAMBattery	DIAG_IPConnections
AIN_5501	DIAG_ModbusStatus
AIN_5502	DIAG_protocolStatus
AIN_5503	DIN_520xDigitalInputs
AIN_5504	DIN_520xInterruptInput
AIN_5521	DIN_520xOptionSwitches
AIN_generic8	DIN_5401
AOUT_5301	DIN_5402
AOUT_5302	DIN_5403
AOUT_5304	DIN_5404
AOUT_generic2	DIN_5405
AOUT_generic4	DIN_5421
CNFG_5904Modem	DIN_generic16
CNFG_clearPortCounters	DIN_generic8
CNFG_clearProtocolCounters	DIN_SP32OptionSwitches
CNFG_IPSettings	DOUT_5401
CNFG_LEDPower	DOUT_5402
CNFG_MTCPIfSettings	DOUT_5406
CNFG_MTCPSettings	DOUT_5407
CNFG_PIDBlock	DOUT_5408
CNFG_portSettings	DOUT_5409
CNFG_protocolExtended	DOUT_5411
CNFG_protocolExtendedEx	DOUT_generic16
CNFG_protocolSettings	DOUT_generic8
CNFG_realTimeClock	SCADAPack_AOUT

CNFG_saveToEEPROM	SCADAPack_lowerIO
CNFG_setSerialPortDTR	SCADAPack_upperIO
CNFG_storeAndForward	SCADAPack_LPIO
CNTR_520xCounterInputs	SCADAPack_100IO
CNTR_5410	SCADAPack_5604IO
CNTR_520xInterruptInput	GFC_4202IO
DIAG_commStatus	GFC_4202IOEx
DIAG_controllerStatus	GFC_4202DSIO
DIAG_LogicStatus	CNFG_DeviceConfig

moduleAddress specifies a unique address for the module. For the valid range for *moduleAddress* refer to the list of modules in the chapter *Register Assignment Reference* of the ***Telepace Ladder Logic Reference and User Manual***. For module addresses com1, com2, com3 or com4 specify 0, 1, 2 or 3 respectively for *moduleAddress*. For module types that have no module address (e.g. CNFG_LEDPower) specify -1 for *moduleAddress.* For SCADAPack module types that have a module address fixed at 0, specify 0 for *moduleAddress.*

startingRegister1 specifies the first register of any unused block of consecutive registers. Refer to the list of modules in the *Register Assignment Reference* for the type and number of registers required for this block. Data read from or written to the module is stored in this block of registers.

If the module type specified has more than one type of I/O, use *startingRegister2*, *startingRegister3*, and *startingRegister4* as applicable. Each start register specifies the first register of an unused block of consecutive registers for each type of input or output on the module. Refer to the list of modules in the *Register Assignment Reference* for the module I/O types. Specify 0 for *startingRegister2*, *startingRegister3*, or *startingRegister4* if not applicable.

Notes

Up to 150 modules may be added to the Register Assignment. If the Register Assignment is full or if an incorrect value is specified for any argument this function returns FALSE; otherwise TRUE is returned.

Output registers specified for certain CNFG type modules are initialized with the current parameter values when the module is added to the Register Assignment (e.g. CNFG_realTimeClock).

Call **clearRegAssignment** first before using the **addRegAssignment** function when creating a new Register Assignment.

Duplicate or overlapping register assignments are not checked for by this function. Overlapping register assignments may result in unpredictable I/O activity.

To obtain error checking that avoids invalid register assignments such as these, use the *Telepace Register Assignment* dialog to build the Register Assignment. Then save the Register Assignment in a Ladder Logic file (e.g. filename.lad) and download it with the C program, or transfer the Register Assignment to the C program using the **clearRegAssignment** and **addRegAssignment** functions.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

clearRegAssignment

Example

```
#include <primitiv.h>

void main(void)
{
       request_resource(IO_SYSTEM);

       /* Create the Register Assignment */
       clearRegAssignment();

       addRegAssignment(SCADAPack_lowerIO, 0, 1,
              10001, 30001, 0);
       addRegAssignment(SCADAPack_AOUT, 0, 40001, 0,
              0, 0);
       addRegAssignment(AOUT_5302, 1, 40003, 0, 0, 0);
       addRegAssignment(DIAG_forceLED, -1, 10017, 0,
              0, 0);
       addRegAssignment(DIAG_controllerStatus, -1,
              30009, 0, 0, 0);
       addRegAssignment(DIAG_protocolStatus, 2, 30010,
              0, 0, 0);

       release_resource(IO_SYSTEM);
}
```

addRegAssignmentEx

Add Register Assignment

Syntax

```
#include <ctools.h>
BOOLEAN addRegAssignmentEx(
       UINT16 moduleType,
       UINT16 moduleAddress,
       UINT16 startingRegister1,
       UINT16 startingRegister2,
       UINT16 startingRegister3,
       UINT16 startingRegister4,
       UINT16 parameters[16]
);
```

Description

The **addRegAssignmentEx** function adds one I/O module to the current Register Assignment of type *moduleType*. The following symbolic constants are valid values for *moduleType*:

AIN_5209Temperature
AIN_5209RAMBattery
AIN_5501
AIN_5502
AIN_5503
AIN_5504
AIN_5505
AIN_5506
AIN_5521
AIN_generic8
AOUT_5301
AOUT_5302
AOUT_5304
AOUT_generic2
AOUT_generic4
CNFG_5904Modem
CNFG_clearPortCounters
CNFG_clearProtocolCounters
CNFG_IPSettings
CNFG_LEDPower
CNFG_modbusIpProtocol
CNFG_MTCPIfSettings
CNFG_MTCPSettings
CNFG_PIDBlock
CNTR_5209CounterInputs
CNTR_5410
CNTR_5209InterruptInput
DIAG_commStatus
DIAG_controllerStatus
DIAG_forceLED
DIAG_IPConnections
DIAG_ModbusStatus
DIAG_protocolStatus
DIN_5209DigitalInputs
DIN_5209InterruptInput
DIN_5401
DIN_5402
DIN_5403
DIN_5404
DIN_5405
DIN_5421
DIN_generic16
DIN_generic8
DOUT_5401
DOUT_5402
DOUT_5406
DOUT_5407
DOUT_5408

CNFG_portSettings	DOUT_5409
CNFG_protocolExtended	DOUT_5411
CNFG_protocolExtendedEx	DOUT_generic16
CNFG_protocolSettings	DOUT_generic8
CNFG_realTimeClock	SCADAPack_AOUT
CNFG_saveToEEPROM	SCADAPack_lowerIO
CNFG_setSerialPortDTR	SCADAPack_upperIO
CNFG_storeAndForward	SCADAPack_LPIO
CNFG_DeviceConfig	SCADAPack_100IO
	SCADAPack_5209IO
	SCADAPack_5606IO

moduleAddress specifies a unique address for the module. For the valid range for *moduleAddress* refer to the list of modules in the chapter *Register Assignment Reference* of the ***Telepace Ladder Logic Reference and User Manual***. For module addresses com1, com2, com3 or com4 specify 0, 1, 2 or 3 respectively for *moduleAddress*. For module address Ethernet1 specify 4 for *moduleAddress*. For module types that have no module address (e.g. CNFG_LEDPower) specify -1 for *moduleAddress.* For SCADAPack module types that have a module address fixed at 0, specify 0 for *moduleAddress.*

startingRegister1 specifies the first register of any unused block of consecutive registers. Refer to the list of modules in the *Register Assignment Reference* for the type and number of registers required for this block. Data read from or written to the module is stored in this block of registers.

If the module type specified has more than one type of I/O, use *startingRegister2*, *startingRegister3*, and *startingRegister4* as applicable. Each start register specifies the first register of an unused block of consecutive registers for each type of input or output on the module. Refer to the list of modules in the *Register Assignment Reference* for the module I/O types. Specify 0 for *startingRegister2*, *startingRegister3*, or *startingRegister4* if not applicable.

Parameters is an array of configuration parameters for the register assignment module. Many modules do not use the parameters and a 0 needs to be specified for the parameters. Use the **addRegAssignment** function to configure these modules. Use parameters with the following modules.

5505 I/O Module: parameters[0] to [3] define the analog input type for the corresponding input. Valid values are:

- 0 = RTD in deg Celsius
- 1 = RTD in deg Fahrenheit
- 2 = RTD in deg Kelvin
- 3 = resistance measurement in ohms.

5505 I/O Module: parameter[4] defines the analog input filter. Valid values are:

- 0 = 0.5 s (minimum)
- 1 = 1 s

- 2 = 2 s
- 3 = 4 s (maximum)

5506 I/O Module: parameters[0] to [7] define the analog input type for the corresponding input. Valid values are:

- 0 = 0 to 5 V input
- 1 = 1 to 5 V input
- 2 = 0 to 20 mA input
- 3 = 4 to 20 mA input

5506 I/O Module: parameter[8] defines the analog input filter. Valid values are:

- 0 = < 3 Hz (maximum filter)
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz (minimum filter)

5506 I/O Module: parameter[9] defines the scan frequency. Valid values are:

- 0 = 60 Hz
- 1 = 50 Hz

5606 I/O Module: parameters[0] to [7] define the analog input type for the corresponding input. Valid values are:

- 0 = 0 to 5 V input
- 1 = 1 to 5 V input
- 2 = 0 to 20 mA input
- 3 = 4 to 20 mA input

5606 I/O Module: parameter[8] defines the analog input filter. Valid values are:

- 0 = < 3 Hz (maximum filter)
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz (minimum filter)

5606 I/O Module: parameter[9] defines the scan frequency. Valid values are:

- 0 = 60 Hz
- 1 = 50 Hz

5606 I/O Module: parameter[10] defines the analog output type. Valid values are:

- 0 = 0 to 20 mA output

- 1 = 4 to 20 mA output

Notes

Up to 150 modules may be added to the Register Assignment. If the Register Assignment is full or if an incorrect value is specified for any argument this function returns FALSE; otherwise TRUE is returned.

Output registers specified for certain CNFG type modules are initialized with the current parameter values when the module is added to the Register Assignment (e.g. CNFG_realTimeClock).

Call clearRegAssignment first before using the addRegAssignmentEx function when creating a new Register Assignment.

Duplicate or overlapping register assignments are not checked for by this function. Overlapping register assignments may result in unpredictable I/O activity.

To obtain error checking that avoids invalid register assignments such as these, use the *Telepace Register Assignment* dialog to build the Register Assignment. Then save the Register Assignment in a Ladder Logic file (e.g. filename.lad) and download it with the C program, or transfer the Register Assignment to the C program using the clearRegAssignment and addRegAssignmentEx functions.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

addRegAssignment, **clearRegAssignment**

ain

Read an Analog Input

Syntax

```
#include <ctools.h>
int ain(unsigned channel);
```

Description

The **ain** function reads from the analog input or output specified by *channel*. Input channels read from the analog input hardware. Output channels read the value output to the channel with the **aout** function.

The valid range for *channel* is 0 to **AIO_MAX**. If an invalid channel is selected, the **ain** function returns **INT_MIN** and the current task's error code is set to **AIO_BADCHAN.** The error code is obtained with the **check_error** function.

The **ain** function normally returns a value in the range –32767 to +32767.

Notes

Use offsets from the symbolic constants AIN_START, AIN_END, AOUT_START and AOUT_END to reference analog channels. The constants make programs more portable and protect against future changes to the analog I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioRead8Ain** directly.

See also

aout, check_error, ioRead8Ain

Example

```
#include <ctools.h>

void main(void)
{
       request_resource(IO_SYSTEM);
       printf("ain(%d)=%d\r\n", 2, ain(2));
       release_resource(IO_SYSTEM);
}
```

aioError

Read Analog I/O Error Flags

Syntax

```
#include <ctools.h>
int aioError(unsigned channel);
```

Description

The **aioError** function reads the I/O error flag for an analog channel.

It returns the error flag for the channel, if the channel number is valid; otherwise it returns INT_MIN. A value of 0 indicates no error occurred. A positive value indicates an error.

Notes

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioRead8Ain** directly.

See Also

aout, check_error, ioRead8Ain

alarmIn

Determine Alarm Time from Elapsed Time

Syntax

```
#include <ctools.h>
ALARM_SETTING alarmIn(unsigned hours, unsigned minutes, unsigned
seconds);
```

Description

The **alarmIn** function calculates the alarm settings to configure a real time clock alarm to occur in *hours*, *minutes* and *seconds* from the current time.

The function returns an ALARM_SETTING structure suitable for passing to the **setClockAlarm** function. The structure specifies an absolute time alarm at the time offset specified by the call to **alarmIn**. Refer to the **Structures and Types** section for a description of the fields in the ALARM_SETTING structure.

Notes

If *second* is greater than 60 seconds, the additional time is rolled into the minutes. If *minute* is greater than 60 minutes, the additional time is rolled into the hours.

If the offset time is greater that one day, then the alarm time will roll over within the current day.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

getClockAlarm, setClockAlarm,

Example

```
#include <ctools.h>

/* --------------------------------------------
   conservePower

   The conservePower function places the
   controller into sleep mode for 10 minutes.
   -------------------------------------------- */
void conservePower(void)
{
       ALARM_SETTING alarm;

       request_resource(IO_SYSTEM);

       /* Alarm in 10 minutes */
       alarm = alarmIn(0, 10, 0);
       setClockAlarm(alarm)

       /* Put controller in low power mode */
       sleep();
```

```
        release_resource(IO_SYSTEM);
}
```

allocate_envelope

Obtain an Envelope from the RTOS

Syntax

```
#include <ctools.h>envelope *allocate_envelope(void);
```

Description

The **allocate_envelope** function obtains an envelope from the operating system. If no envelope is available, the task is blocked until one becomes available.

The **allocate_envelope** function returns a pointer to the envelope.

Notes

Envelopes are used to send messages between tasks. The RTOS allocates envelopes from a pool of free envelopes. It returns envelopes to the pool when they are de-allocated.

An application program needs to ensure that unneeded envelopes are de-allocated. Envelopes may be reused.

See Also

deallocate_envelope

Example

```
#include <ctools.h>
extern unsigned other_task_id;

void task1(void)
{
	envelope *letter;

	/* send a message to another task */
	/* assume it will deallocate the envelope */

	letter = allocate_envelope();
	letter->destination = other_task_id;
	letter->type = MSG_DATA;
	letter->data = 5;
	send_message(letter);

	/* receive a message from any other task */

	letter = receive_message();
	/* ... process the data here */
	deallocate_envelope(letter);

	/* ... the rest of the task */
}
```

aout

Write to Analog Output

Syntax

```
#include <ctools.h>
int aout(unsigned channel, int value);
```

Description

The **aout** function writes *value* to the analog output specified by *channel*. The range for *channel* is **AOUT_START** to **AOUT_END** inclusive. The range for *value* is -32767 to 32767.

aout returns the value written to the hardware, or -1 if the channel is not an analog output.

Notes

The *value* output may be limited by the analog output module.

Use offsets from the symbolic constants AIN_START, AIN_END, AOUT_START and AOUT_END to reference analog channels. The constants make programs more portable and protect against future changes to the analog I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioWrite4Aout** directly.

See Also

addRegAssignment, ioWrite4Aout

Example

```
#include <ctools.h>
void main(void)
{
	int value;

	/* ramp output from zero to full scale */
	for (value = 0; value < 32767; value++)
	{
		request_resource(IO_SYSTEM);
		aout(AOUT_START, value);
		release_resource(IO_SYSTEM);
	}
}
```

auto_pid

Execute a PID Block Automatically

Syntax

```
#include <ctools.h>
void auto_pid(unsigned block, unsigned period);
```

Description

The **auto_pid** routine configures a PID control block to execute automatically at the specified period. *period* is measured in 0.1 second increments. *block* needs to be in the range 0 to **PID_BLOCKS** – 1.

Setting the period to 0 stops execution of the control block.

Notes

See the ***Telepace PID Controllers Reference Manual*** for a detailed description of PID control.

The control block needs to be configured properly before it is engaged, or indeterminate operation may result.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

set_pid, clear_pid

check_error

Get Error Code for Current Task

Syntax

```
#include <ctools.h>
int check_error(void);
```

Description

The **check_error** function returns the error code for the current task. The error code is set by various I/O routines, when errors occur. A separate error code is maintained for each task.

Notes

Some routines in the standard C library, return errors in the global variable **errno**. This variable is not unique to a task, and may be modified by another task, before it can be read.

See Also

report_error

checksum

Calculate a Checksum

Syntax

```
#include <ctools.h>
unsigned checksum(unsigned char *start, unsigned char *end,
unsigned algorithm);
```

Description

The **checksum** function calculates a checksum on memory. The memory starts at the byte pointed to by *start*, and ends with the byte pointed to by *end*. The *algorithm* may be one of:

ADDITIVE	16 bit byte-wise sum
CRC_16	CRC-16 polynomial checksum
CRC_CCITT	CRC-CCITT polynomial checksum
BYTE_EOR	8 bit byte-wise exclusive OR

The CRC checksums use the **crc_reverse** function.

See Also

crc_reverse

Example

This function displays two types of checksums.

```
#include <ctools.h>

void checksumExample(void)
{
	char str[] = "This is a test";
	unsigned sum;

	/* Display additive checksum */
	sum = checksum(str, str+strlen(str), ADDITIVE);
	printf("Additive checksum: %u\r\n", sum);

	/* Display CRC-16 checksum */
	sum = checksum(str, str+strlen(str), CRC_16);
	printf("CRC-16 checksum: %u\r\n", sum);
}
```

checkSFTranslationTable

Test for Store and Forward Configuration Errors

Syntax

#include <ctools.h>

struct SFTranslationStatus checkSFTranslationTable(void);

Description

The **checkSFTranslationTable** function checks all entries in the address translation table for validity. It detects the following errors:

The function returns a *SFTranslationStatus* structure. Refer to the **Structures and Types** section for a description of the fields in the *SFTranslationStatus* structure. The *code* field of the structure is set to one of the following. If there is an error, the *index* field is set to the location of the translation that is not valid.

Result code	Meaning
SF_VALID	All translations are valid
SF_NO_TRANSLATION	The entry defines re-transmission of the same message on the same port
SF_PORT_OUT_OF_RANGE	One or both of the serial port indexes is not valid
SF_STATION_OUT_OF_RANGE	One or both of the stations is not valid

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

See Also

getSFTranslation, setSFTranslation, checkSFTranslationTable

Example

See the example for the **setSFTranslation** function.

clearAllForcing

Clear All Forcing

Syntax

```
#include <ctools.h>
void clearAllForcing(void);
```

Description

The **clearAllForcing** function removes all forcing conditions from all I/O database registers.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

setForceFlag, overrideDbase

clear_errors

Clear Serial Port Error Counters

Syntax

```
#include <ctools.h>
void clear_errors(FILE *stream);
```

Description

The **clear_errors** function clears the serial port error counters for the serial port specified by *stream*. If *stream* does not point to a valid serial port the function has no effect.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

get_status

clear_pid

Clear PID Block Variables

Syntax

```
#include <ctools.h>
void clear_pid(unsigned block);
```

Description

The **clear_pid** routine sets all variables in the specified control block to 0. **clear_pid** is normally used as the first step of control block configuration. *block* needs to be in the range 0 to **PID_BLOCKS** – 1.

Notes

See the ***Telepace PID Controllers Reference Manual*** for a detailed description of PID control.

Values stored in PID blocks are not initialized when a program is run, and are guaranteed to retain their values during power failures and program loading. PID block variables need to be initialized by the user program.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

auto_pid

Clear Protocol Counters

Syntax

```
#include <ctools.h>
void clear_protocol_status(FILE *stream);
```

Description

The **clear_protocol_status** function clears the error and message counters for the serial port specified by *stream*. If *stream* does not point to a valid serial port the function has no effect.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

get_protocol_status

clearRegAssignment

Clear Register Assignment

Syntax

```
#include <ctools.h>
void clearRegAssignment(void);
```

Description

The **clearRegAssignment** function erases the current Register Assignment. Call this function first before using the **addRegAssignment** function to create a new Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

addRegAssignment

Example

See example for **addRegAssignment**.

clearSFTranslationTable

Clear Store and Forward Translation Configuration

Syntax

```
#include <ctools.h>
void clearSFTranslationTable(void);
```

Description

The **clearSFTranslationTable** function clears all entries in the store and forward translation table.

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

getSFTranslation, setSFTranslation, checkSFTranslationTable

Example

See the example for the **setSFTranslation** function.

clearStatusBit

Clear Bits in Controller Status Code

Syntax

```
#include <ctools.h>
unsigned clearStatusBit(unsigned bitMask);
```

Description

The **clearStatusBit** function clears the bits indicated by *bitMask* in the controller status code. When the status code is non-zero, the STAT LED blinks a binary sequence corresponding to the code. If *code* is zero, the STAT LED turns off.

The function returns the value of the status register.

Notes

The status output opens if *code* is non-zero. Refer to the ***System Hardware Manual*** for more information.

The binary sequence consists of short and long flashes of the error LED. A binary zero is indicated by a short flash of 1/10th of a second. A longer flash of approximately 1/2 of a second indicates a binary one. The least significant digit is output first. As few bits as possible are displayed – leading zeros are ignored. There is a two-second delay between repetitions.

The STAT LED is the LED located on the top left hand corner of the 5203 or 5204 controller board.

Bits 0 and 1 of the status code are used by the Register Assignment.

See Also

setStatusBit, setStatus, getStatusBit

clear_tx

Clear Serial Port Transmit Buffer

Syntax

```
#include <ctools.h>
void clear_tx(FILE *stream);
```

Description

The **clear_tx** function clears the transmit buffer for the serial port specified by *stream*. If *stream* does not point to a valid serial port the function has no effect.

See Also

get_status

configurationRegisterMapping

Enable or disable mapping of device configuration registers.

Syntax

```
#include <ctools.h>
void configurationRegisterMapping(
      BOOLEAN enabled
);
```

Description

This function enables or disables mapping of device configuration registers. These registers are located at a fixed location in the input register area.

enabled selects if the registers are mapped. Valid values are TRUE and FALSE. Selecting FALSE hide the configuration data but does not change it.

See Also

configurationSetApplicationID

configurationSetApplicationID

Set an application ID.

Syntax

```
#include <ctools.h>
BOOLEAN configurationSetApplicationID(
        UINT16 applicationType,
        UINT16 action,
        UINT16 companyID,
        UINT16 application,
        UINT16 version
);
```

Description

This function stores or removes an application ID in the device configuration data. The device configuration appears in Modbus registers if the register mapping is enabled.

applicationType specifies the type of application. It is one of DCAT_LOGIC1, DCAT_LOGIC2, or DCAT_C.

- DCAT_LOGIC1: Device configuration application type is the first logic application.
- DCAT_LOGIC2: Device configuration application type is the second logic application.
- DCAT_C: Device configuration application type is a C application.

If DCAT_C is used, the application ID is added to the table of C applications. The applications don’t appear in any fixed order in the C application table.

action specifies if the ID is to be added or removed. Valid values are DCA_ADD and DCA_REMOVE.

- DCA_ADD: attempting to add a duplicate value (matching companyID, application, and version) will result in only one entry in the table. The function will return TRUE (indicating the data is in the table).
- DCA_REMOVE: For logic applications the ID will be removed unconditionally. For C applications, the ID will be removed if it is found in the table (matching companyID, application, and version).

companyID specifies your company. Contact Control Microsystems to obtain a company ID. 0 indicates an unused entry.

application specifies your application. Valid values are 0 to 65535. You need to maintain unique values for your company.

version is the version of your application in the format major * 100 + minor. Valid values are 0 to 65535.

The function returns TRUE if the action was successful, and FALSE if an error occurred.

Register Mapping

The Device configuration is stored in Modbus input (3xxxx) registers as shown below. The registers are read with standard Modbus commands. These registers cannot be written to. Device configuration registers used fixed addresses. This facilitates identifying the applications in a standard manner.

The Device configuration registers can be enabled or disabled by entering a 0 or 1 in the Start Register. They are disabled until enabled by a logic application. This provides compatibility with controllers that have already used these registers for other purposes.

The application IDs are cleared on every controller reset. Applications need to run and set the application ID for it to be valid.

These data types are used.

Data Type	Description
uint	Unsigned 16–bit integer
uchar	Unsigned 8–bit character
type[n]	n–element array of specified data *type*

The following information is stored in the device configuration. 2 logic application identifiers are provided for compatibility with SCADAPack ES/ER controllers that provide 2 IEC 61131-3 applications. The second logic application identifier is not used with other controllers. 32 application identifiers are provided to accommodate C applications in SCADAPack 330/350 controllers.

Register	Data Type	Description
39800	uchar[8]	Controller ID (padded with nulls = 0), first byte in lowest register, one byte per register.
39808	uint	Firmware version (major*100 + minor)
39809	uint	Firmware version build number (if applicable)
39810	uint[3]	Logic application 1 identifier (see format below)
39813	uint[3]	Logic application 2 identifier (see format below)
39816	uint	Number of applications identifiers used (0 to 32) Identifiers are listed sequentially starting with identifier 1. Unused identifiers will return 0.
39817	uint[3]	Application identifier 1 (see format below)
39820	uint[3]	Application identifier 2 (see format below)
39823	uint[3]	Application identifier 3 (see format below)
39826	uint[3]	Application identifier 4 (see format below)
39829	uint[3]	Application identifier 5 (see format below)
39832	uint[3]	Application identifier 6 (see format below)
39835	uint[3]	Application identifier 7 (see format below)
39838	uint[3]	Application identifier 8 (see format below)

Register	Data Type	Description
39841	uint[3]	Application identifier 9 (see format below)
39844	uint[3]	Application identifier 10 (see format below)
39847	uint[3]	Application identifier 11 (see format below)
39850	uint[3]	Application identifier 12 (see format below)
39853	uint[3]	Application identifier 13 (see format below)
39856	uint[3]	Application identifier 14 (see format below)
39859	uint[3]	Application identifier 15 (see format below)
39862	uint[3]	Application identifier 16 (see format below)
39865	uint[3]	Application identifier 17 (see format below)
39868	uint[3]	Application identifier 18 (see format below)
39871	uint[3]	Application identifier 19 (see format below)
39874	uint[3]	Application identifier 20 (see format below)
39877	uint[3]	Application identifier 21 (see format below)
39880	uint[3]	Application identifier 22 (see format below)
39883	uint[3]	Application identifier 23 (see format below)
39886	uint[3]	Application identifier 24 (see format below)
39889	uint[3]	Application identifier 25 (see format below)
39892	uint[3]	Application identifier 26 (see format below)
39895	uint[3]	Application identifier 27 (see format below)
39898	uint[3]	Application identifier 28 (see format below)
39901	uint[3]	Application identifier 29 (see format below)
39904	uint[3]	Application identifier 30 (see format below)
39907	uint[3]	Application identifier 31 (see format below)
39910	uint[3]	Application identifier 32 (see format below)
39913 to 39999		Reserved for future expansion

Application Identifier

The application identifier is formatted as follows.

Data Type	Description
uint	Company ID (see below)
uint	Application number (0 to 65535)
uint	Application version (major*100 + minor)

Company Identifier

Control Microsystems will maintain a list of company identifiers to ensure the company ID is unique. Contact the technical support department.

Company ID 0 indicates an identifier is unused.

See Also

configurationRegisterMapping

Notes

Application IDs for C programs are not automatically removed. A task exit handler can be used to remove the ID when the C application is ended.

Application IDs are cleared when the controller is reset.

counter

Read Counter Input Module

Syntax

```
#include <ctools.h>
long counter(unsigned counter);
```

Description

The **counter** function reads data from the counter input specified by *channel*. If the channel number is not valid a COUNTER_BADCOUNTER error is reported for the current task. The value returned by **counter** is not valid.

Notes

Refer to the ***Telepace Ladder Logic User Manual*** for an explanation of counter input channel assignments.

Use offsets from the symbolic constants COUNTER_START and COUNTER_END to reference counter channels. The constants make programs more portable and protect against future changes to the counter input channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioRead4Counter** directly.

See Also

counterError, check_error, request_resource, release_resource, ioRead4Counter

counterError

Read Counter Input Error Flag

Syntax

```
#include <ctools.h>
long counterError(unsigned counter);
```

Description

The **counterError** function returns the I/O error flag for a counter channel. It returns TRUE if an error occurred and FALSE if no occurred on the last read of the input module.

If the channel number is not valid a COUNTER_BADCOUNTER error is reported for the current task. The value returned is not valid.

Notes

Refer to the ***Telepace Ladder Logic User Manual*** for a explanation of counter input channel assignments.

Use offsets from the symbolic constants COUNTER_START and COUNTER_END to reference counter channels. The constants make programs more portable and protect against future changes to the counter input channel numbering.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioRead4Counter** directly.

See Also

counter, check_error, ioRead4Counter

crc_reverse

Calculate a CRC Checksum

Syntax

```
#include <ctools.h>
unsigned crc_reverse(unsigned char *start, unsigned char *end,
unsigned poly, unsigned initial);
```

Description

The **crc_reverse** function calculates a CRC type checksum on memory using the reverse algorithm. The memory starts at the byte pointed to by *start*, and ends with the byte pointed to by *end*. The generator polynomial is specified by *poly*. *poly* may be any value, but needs to be carefully chosen to ensure good error detection. The checksum accumulator is set to *initial* before the calculation is started.

Notes

The reverse algorithm is named for the direction bits are shifted. In the reverse algorithm, bits are shifted towards the least significant bit. This produces different checksums than the classical, or forward algorithm, using the same polynomials.

See Also

checksum

create_task

Create a New Task

Syntax

```
#include <ctools.h>
int create_task(void *function, unsigned priority, unsigned type,
unsigned stack);
```

Description

The **create_task** function allocates stack space for a task and places the task on the ready queue. *function* specifies the start address of the routine to be executed. The task will execute immediately if its priority is higher than the current task.

priority is an execution priority between 1 and 4 for the created task. The 4 task priority levels aid in scheduling task execution.

type specifies if the task is ended when an application program is stopped. Valid values for *type* are:

SYSTEM	system tasks are not terminated when the program stops
APPLICATION	application tasks terminate when the program stops

It is recommended that only **APPLICATION** type tasks be created.

The *stack* parameter specifies how many stack blocks are allocated for the task. Each stack block is 256 bytes.

The **create_task** function returns the task ID (TID) of the task created. If an error occurs, -1 is returned.

Notes

Refer to the **Real Time Operating System** section for more information on tasks.

The **main** task and the Ladder Logic and I/O scanning task have a priority of 1. If the created task is continuously running processing code, create the task with a priority of 1 and call **release_processor** periodically; otherwise the remaining priority 1 tasks will be blocked from executing.

For tasks such as a protocol handler, that wait for an event using the **wait_event** or **receive_message** function, a priority greater than 1 may be selected without blocking other lower priority tasks.

The number of stack blocks required depends on the functions called within the task, and the size of local variables created. Most tasks require 2 stack blocks. If any of the **printf** functions are used, then at least 4 stack blocks are required. Add local variable usage to these limits, if large local arrays or structures are created. Large structures and arrays are usually best handled as static global variables within the task source file. (The variables are global to all functions in the task, but cannot be seen by functions in other files.)

Additional stack space may be made available by disabling unused protocol tasks. See the section **Program Development** or the set_protocol() function for more information.

See Also

end_task

Example

```
#include <ctools.h>

#define       TIME_TO_PRINT       20

void task1(void)
{
       int a, b;

       while (TRUE)
       {
              /* body of task 1 loop - processing I/O */

              request_resource(IO_SYSTEM);
              a = dbase(MODBUS, 30001);
              b = dbase(MODBUS, 30002);
              setdbase(MODBUS, 40020, a * b);
              release_resource(IO_SYSTEM);

              /* Allow other tasks to execute */
              release_processor();
       }
}

void task2(void)
{
       while(TRUE)
       {
              /* body of task 2 loop - event handler */
              wait_event(TIME_TO_PRINT);
              printf("It's time for a coffee break\r\n");
       }
}
/* --------------------------------------------
   The shutdown function stops the signalling
   of TIME_TO_PRINT events when application is
       stopped.
   -------------------------------------------- */
void shutdown(void)
{
       endTimedEvent(TIME_TO_PRINT);
}
void main(void)
{
       TASKINFO taskStatus;

       /* continuos processing task at priority 1 */
```

```
        create_task(task1, 1, APPLICATION, 2);

        /* event handler needs larger stack for printf function */
        create_task(task2, 3, APPLICATION, 4);

        /* set up task exit handler to stop
           signalling of events when this task ends */
        taskStatus = getTaskInfo(0);
        installExitHandler(taskStatus.taskID, shutdown);

        /* start timed event to occur every 10 sec */
        startTimedEvent(TIME_TO_PRINT, 100);

        interval(0, 10);
        while(TRUE)
        {
                /* body of main task loop */
                /* other processing code */
                /* Allow other tasks to execute */
                release_processor();
        }
}
```

databaseRead

Read Value from I/O Database

Syntax

```
#include <ctools.h>
BOOLEAN databaseRead(UINT16 type, UINT16 address, INT16* value)
```

Description

The **databaseRead** function reads a value from the database. The value is written to the variable pointed to by value. The variable is not changed if type and address are not valid.

The function has three parameters. type specifies the method of addressing the database. Valid values are MODBUS and LINEAR. address specifies the location in the database. value is a pointer to a variable to hold the result.

The function returns TRUE if the specified address is valid and FALSE if the register does not exist.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

databaseWrite

Example

```
#include <ctools.h>
void main(void)
{
       INT16 value;
       BOOLEAN status;

       request_resource(IO_SYSTEM);

       /* Read Modbus status input point */
       status = databaseRead(MODBUS, 10001, &value);

       /* Read 16 bit register */
       status = databaseRead(LINEAR, 3020, &value);

       release_resource(IO_SYSTEM);
}
```

databaseWrite

Write Value to I/O Database

Syntax

```
#include <ctools.h>
BOOLEAN databaseWrite(UINT16 type, UINT16 address, INT16 value)
```

Description

The **databaseWrite** function writes value to the I/O database.

The function has three parameters. type specifies the method of addressing the database. Valid values are MODBUS and LINEAR. address specifies the location in the database. value is the data to write.

The function returns TRUE if the value was written. The function returns FALSE if

- the type is invalid
- the address is not valid for the controller
- the address is read only on the SCADAPack 4202 controller (some registers in the range 40001 to 40499).
- the data is not valid for the address on the SCADAPack 4202 controller (some registers in the range 40001 to 40499).
- the hardware write protect is installed on the SCADAPack 4202 controller (registers in the range 40001 to 40499).
- the flow computer is running on the SCADAPack 4202 controller (registers in the range 40001 to 40499).

Notes

When writing to LINEAR digital addresses, *value* is a bit mask which writes data to 16 1-bit registers at once. If any of these 1-bit registers is invalid, only the valid registers are written and FALSE is returned.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

databaseRead

Example

```
#include <ctools.h>

void main(void)
{
	BOOLEAN status;
	request_resource(IO_SYSTEM);

	status = databaseWrite(MODBUS, 40001, 102);
```

```
        /* Turn ON the first 16 coils */
        status = databaseWrite(LINEAR, 0, 255);

        /* Write to a 16 bit register */
        status = databaseWrite(LINEAR, 3020, 240);

        release_resource(IO_SYSTEM);
}
```

datalogCreate

Create Data Log Function

Syntax

```
#include <ctools.h>
DATALOG_STATUS datalogCreate(
       UINT16 logID,
       DATALOG_CONFIGURATION * pLogConfiguration
       );
```

Description

This function creates a data log with the specified configuration. The data log is created in the data log memory space.

The function has two parameters. logID specifies the data log to be created. The valid range is 0 to 15. pLogConfiguration points to a structure with the configuration for the data log.

The function returns the status of the operation.

Notes

The configuration of an existing data log cannot be changed. The log needs to be deleted and recreated to change the configuration.

Data logs are stored in memory from a pool for all data logs. If there is insufficient memory the creation operation fails. The function returns DLS_NOMEMORY.

If the data log already exists the creation operation fails. The function returns DLS_EXISTS.

If the log ID is not valid the creation operation fails. The function returns DLS_BADID.

If the configuration is not valid the creation operation fails. The function returns DLS_BADCONFIG.

See Also

datalogDelete datalogSettings

Example

```
/*-----------------------------------------------
  The following code shows how to create a
  data log and how to write one record into it.
  -----------------------------------------------*/
#include "ctools.h"
/*---------------------------------
  Structure used only to copy one
  record into data log
---------------------------------*/
struct dataRecord
```

```
{
       UINT16       value1;
       int          value2;
       double       value3;
       float  value4;
       float  value5;
};
int logID;
/*-----------------------------------
  Declare a structure for the log
-----------------------------------*/
DATALOG_CONFIGURATION dLogConfig;
/*-----------------------------------
  Declare a struture to hold the
  data that will be copied in log
-----------------------------------*/
struct dataRecord data;
/*-----------------------
  Function declaration
-----------------------*/
void ConfigureLog(void);
void InitRecord(void);

void main(void)
{
       ConfigureLog();            /* function call to cofigure log
*/
       InitRecord();

       if(datalogCreate(logID, &dLogConfig) == DLS_CREATED)
       {
              /* Start writing records in log */
              if( datalogWrite(logID, (UINT16 *)&data) )
              {
                     /* one record was written in data log */
              }
       }
}

/* Log configuration */
void ConfigureLog(void)
{
       /* Assign a number to the data log */
       logID = 10;

       /* Fill in the log configuration structure */
       dLogConfig.records = 200;
       dLogConfig.fields = 5;
       dLogConfig.typesOfFields[0] = DLV_UINT16;
       dLogConfig.typesOfFields[1] = DLV_INT32;
       dLogConfig.typesOfFields[2] = DLV_DOUBLE;
       dLogConfig.typesOfFields[3] = DLV_FLOAT;
       dLogConfig.typesOfFields[4] = DLV_FLOAT;
}

/* One record initialization */
```

```
void InitRecord(void)
{
	/* Assign some data for the log */
	data.value1 = 100;
	data.value2 = 200;
	data.value3 = 30000;
	data.value4 = 40.3;
	data.value5 = 50.75;
}
```

datalogDelete

Delete Data Log Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogDelete(
       UINT16 logID
       );
```

Description

This function destroys the specified data log. The memory used by the data log is returned to the freed.

The function has one parameter. logID specifies the data log to be deleted. The valid range is 0 to 15.

The function returns TRUE if the data log was deleted. The function returns FALSE if the log ID is not valid or if the log had not been created.

See Also

datalogCreate

Example

```
/* The following code shows the only way to
   change the configuration of an existing log
   is to delete the log and recreate the data
   log                                         */

#include <ctools.h>

int logID;

/* Declare a structure for the log */
DATALOG_CONFIGURATION dLogConfig;

/* Select logID #10 */
logID = 10;

/* Read the configuration of logID #10 */
if( datalogSettings( logID,  &dLogConfig ) )
{
  if(dLogConfig.typesOfFields[0] == DLV_INT16)
  {
    /* Wrong type.  Delete whole log and start from scratch */
    if(datalogDelete(logID) )
    {
      /* Re-enter the log configuration */
       dLogConfig.records = 200;
       dLogConfig.fields = 5;
       dLogConfig.typesOfFields[0] = DLV_UINT16;
       dLogConfig.typesOfFields[1] = DLV_INT32;
```

```
         dLogConfig.typesOfFields[2] = DLV_DOUBLE;
         dLogConfig.typesOfFields[3] = DLV_FLOAT;
         dLogConfig.typesOfFields[4] = DLV_FLOAT;
        datalogCreate(logID, &dLogConfig);
      }
      else
      {
        /* could not delete log */
            }
    }
  }
  else
  {
    /* Could not read settings */
              }
```

datalogPurge

Purge Data Log Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogPurge(
       UINT16 logID,
       BOOLEAN purgeAll,
       UINT32 sequenceNumber
       );
```

Description

This function removes records from a data log. The function can remove all the records, or a group of records starting with the oldest in the log.

The function has three parameters. logID specifies the data log. The valid range is 0 to 15. If purgeAll is TRUE, all records are removed, otherwise the oldest records are removed. sequenceNumber specifies the sequence number of the most recent record to remove. All records up to and including this record are removed. This parameter is ignored if purgeAll is TRUE.

The function returns TRUE if the operation succeeds. The function returns FALSE if the log ID is invalid, if the log has not been created, or if the sequence number cannot be found in the log.

Notes

Purging the oldest records in the log is usually done after reading the log. The sequence number used is that of the last record read from the log. This removes the records that have been read and leaves any records added since the records were read.

If the sequence number specifies a record that is not in the log, no records are removed.

See Also

datalogReadStart datalogReadNext datalogWrite

Example

```
#include <ctools.h>

int logID, sequenceNumber;

/* Declare flag to purge entire of data log or part of it */
BOOLEAN purgeAll;

/* Which data log to purge? */
logID = 10;

/* Set flag to purge only part of data log */
```

```
purgeAll = FALSE;

/* How many of the oldest records to purge */
sequenceNumber = 150;

if( datalogPurge(logID, purgeAll, sequenceNumber) )
{
       /* Successful at purging the first 150 records of log */
       /* Start writing records again */
}

/* To purge the entire data log, simply set flag to TRUE */
purgeAll = TRUE;

/* Call up function with same parameters */
if( datalogPurge(logID, purgeAll, sequenceNumber) )
{
       /* Successful at purging the entire data log */
       /* Start writing records again */
}
```

datalogReadNext

Read Data Log Next Function

This function returns the next record in the data log.

Syntax

```
#include <ctools.h>
BOOLEAN datalogReadNext(
       UINT16 logID,
       UINT32 sequenceNumber,
       UINT32 * pSequenceNumber,
       UINT32 * pNextSequenceNumber,
       UINT16 * pData
       );
```

Description

This function reads the next record from the data log starting at the specified sequence number. The function returns the record with the specified sequence number if it is present in the log. If the record no longer exists it returns the next record in the log.

The function has five parameters. logID specifies the data log. The valid range is 0 to 15. sequenceNumber is sequence number of the record to be read. pSequenceNumber is a pointer to a variable to hold the sequence number of the record read. pNextSequenceNumber is a pointer to a variable to hold the sequence number of the next record in the log. This is normally used for the next call to this function. pData is a pointer to memory to hold the data read from the log.

The function returns TRUE if a record is read from the log. The function returns FALSE if the log ID is not valid, if the log has not been created or if there are no more records in the log.

Notes

Use the datalogReadStart function to obtain the sequence number of the oldest record in the data log.

The pData parameter needs to point to memory of sufficient size to hold all the data in a record.

It is normally necessary to call this function until it returns FALSE in order to read all the data from the log. This accommodates cases where data is added to the log while it is being read.

If data is read from the log at a slower rate than it is logged, it is possible that the sequence numbers of the records read will not be sequential. This indicates that records were overwritten between calls to read data.

The sequence number rolls over after reaching its maximum value.

See Also

datalogReadStart datalogPurge datalogWrite

Example

See the example for **datalogReadStart**.

datalogReadStart

Read Data Log Start Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogReadStart(
       UINT16 logID,
       UINT32 * pSequenceNumber
       );
```

Description

This function returns the sequence number of the record at the start of the data log. This is the oldest record in the log.

The function has two parameters. logID specifies the data log. The valid range is 0 to 15. pSequenceNumber is a pointer to a variable to hold the sequence number.

The function returns TRUE if the operation succeeded. The function returns FALSE if the log ID is not valid or if the log has not been created.

Notes

Use the datalogReadNext function to read records from the log.

The function will return a sequence number even if the log is empty. In this case the next call to datalogReadNext will return no data.

See Also

datalogReadNext datalogPurge datalogWrite

Example

```
/***********************************************
  The following code shows how to read records
  from data log.
***********************************************/

#include "ctools.h"
#include <stdlib.h>

UINT16 recordSize,
       logID,
       *pData; /* Pointer to memory to hold data read from log. */

UINT32 sequenceNumber,/* Sequence number of record to be read. */
        nextSequenceNumber; /* Sequence number of next record.  */

void main(void)
{
  /* Select data log #10 */
  logID = 10;
```

```
  /* Find first record in data log #10 and store
    its sequence number into sequenceNumber */

  if( datalogReadStart(logID, &sequenceNumber) )
    {
       /* Get the size of this record */
       if( datalogRecordSize(logID, &recordSize) )
         {
         /* Allocate memory of size recordSize */
         pData = (UINT16 *) malloc(recordSize);

        /* Read all records from data log #10. */
           while( datalogReadNext(logID, sequenceNumber,
&sequenceNumber, &nextSequenceNumber, pData) )
               {
                     /* Use pData and its contents.
                        Set next sequence number of record to be
read. */
                     sequenceNumber =  nextSequenceNumber;
                }
           }
      }
  }
```

datalogRecordSize

Data Log Record Size Function

Syntax

```
#include < ctools.h >
BOOLEAN datalogRecordSize(
       UINT16 logID,
       UINT16 * pRecordSize;
       );
```

Description

This function returns the size of a record for the specified data log. The log needs to have been previously created with the datalogCreate function.

The function has two parameters. logID specifies the data log. The valid range is 0 to 15. pRecordSize points to a variable that will hold the size of a record in the log.

The function returns TRUE if the operation succeeded. The function returns FALSE if the log ID is invalid or if the data log does not exist.

Notes

This function is useful in determining how much memory needs to be allocated for a call to datalogReadNext or datalogWrite.

See Also

datalogSettings

Example

See the example for **datalogReadStart**.

datalogSettings

Data Log Settings Function

Syntax

```
#include < ctools.h >
BOOLEAN datalogSettings(
       UINT16 logID,
       DATALOG_CONFIGURATION * pLogConfiguration
       );
```

Description

This function reads the configuration of the specified data log. The log needs to have been previously created with the datalogCreate function.

The function has two parameters. logID specifies the data log. The valid range is 0 to 15. pLogConfiguration points to a structure that will hold the data log configuration.

The function returns TRUE if the operation succeeded. The function returns FALSE if the log ID is invalid or if the data log does not exist.

Notes

The configuration of an existing data log cannot be changed. The log needs to be deleted and recreated to change the configuration.

See Also

datalogRecordSize

Example

See example for **datalogDelete**.

datalogWrite

Write Data Log Function

Syntax

```
#include <ctools.h>
BOOLEAN datalogWrite(
       UINT16 logID,
       UINT16 * pData
       );
```

Description

This function writes a record to the specified data log. The log needs to have been previously created with the datalogCreate function.

The function has two parameters. logID specifies the data log. The valid range is 0 to 15. pData is a pointer to the data to be written to the log. The amount of data copied using the pointer is determined by the configuration of the data log.

The function returns TRUE if the data is added to the log. The function returns FALSE if the log ID is not valid or if the log does not exist.

Notes

Refer to the datalogCreate function for details on the configuration of the data log.

If the data log is full, then the oldest record in the log is replaced with this record.

See Also

datalogReadStart datalogReadNext datalogPurge

Example

See the example for **datalogDelete**.

dbase

Read Value from I/O Database

Syntax

#include <ctools.h>

int dbase(unsigned *type*, unsigned *address*);

Description

The **dbase** function reads a value from the I/O database. *type* specifies the method of addressing the database. *address* specifies the location in the database. The table below shows the valid address types and ranges

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

Refer to the *I/O Database and Register Assignment* chapter for more information.

If the specified register is currently forced, **dbase** returns the forced value for the register.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

setdbase

Example

```
#include <ctools.h>

void main(void)
{
	int a;

	request_resource(IO_SYSTEM);

	/* Read Modbus status input point */
	a = dbase(MODBUS, 10001);

	/* Read 16 bit register */
```

```
        a = dbase(LINEAR, 3020);

        /* Read 16 bit register beginning at first
        status register */
        a = dbase(LINEAR, START_STATUS);

        /* Read 6th input register */
        a = dbase(LINEAR, START_INPUT + 5);

        release_resource(IO_SYSTEM);
}
```

deallocate_envelope

Return Envelope to the RTOS

Syntax

#include <ctools.h>

void deallocate_envelope(envelope **penv*);

Description

The **deallocate_envelope** function returns the envelope pointed to by *penv* to the pool of free envelopes maintained by the operating system.

See Also

allocate_envelope

Example

See the example for the **allocate_envelope** function.

din

Read Digital I/O

Syntax

#include <ctools.h>

int din(unsigned *channel*);

Description

The **din** function reads the value of a digital input or output channel. Reading an input channel returns data read from a digital input module. Reading an output channel returns the last value written to the output module.

The **din** function returns a value corresponding to the sum of the binary states of all 8 bits of the channel.

Notes

The **din** function reads the status of digital input signals, and digital output modules.

The **din** function may be used to read the current values in the I/O disable, forced status and I/O form tables, and I/O type tables.

Use offsets from the symbolic constants DIN_START, DIN_END, DOUT_START, DOUT_END, EXTENDED_DIN_START, EXTENDED_DIN_END, EXTENDED_DOUT_START and EXTENDED_DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioRead8Din** directly.

See Also

pulse, timeout, turnon, turnoff, on, off

Example

This program displays the first 8 digital inputs in binary.

```
#include <ctools.h>

void main(void)
{
	int loop, value;

	/* Read the first digital input channel */
	request_resource(IO_SYSTEM);
	value = din(DIN_START);
```

```
        release_resource(IO_SYSTEM);

        printf("Channel =");

        /* For each bit in the channel */
        for(loop = 8; loop; loop--)
        {
                putchar((value & 0x80) ? '1' :'0')

                /* Select the next bit */
                value <<= 1;
        }
        puts( "\r\n" );
}
```

dnpInstallConnectionHandler

Configures the connection handler for DNP.

Syntax

#include <ctools.h>

void dnpInstallConnectionHandler(void (* function)
(DNP_CONNECTION_EVENT event));

Description

This function installs a handler that will permit user-defined actions to occur when DNP requires a connection, message confirmation is received, or a timeout occurs.

function is a pointer to the handler function. If function is NULL the handler is disabled.

The function has no return value.

Notes

The handler function needs to process the event and return immediately. If the required action involves waiting this needs to be done outside of the handler function. See the example below for one possible implementation.

The application needs to disable the handler when the application ends. This prevents the protocol driver from calling the handler while the application is stopped. Call the dnpInstallConnectionHandler with a NULL pointer. The usual method is to create a task exit handler function to do this. See the example below for details.

The handler function has one parameter.

- event is DNP event that has occurred. It may be one of DNP_CONNECTION_REQUIRED, DNP_MESSAGE_COMPLETE, or DNP_MESSAGE_TIMEOUT. See the structure definition for the meaning of these events.

The handler function has no return value.

By default no connection handler is installed and no special steps are taken when DNP requires a connection, receives a message confirmation, or a timeout occurs.

See Also

dnpConnectionEvent

Example

This example shows how a C application can handle the events and inform a logic application of the events. The logic application is responsible for making and ending the dial-up connection.

The program uses the following registers.

- 10001 turns on when a connection is requested by DNP for unsolicited reporting.
- 10002 turns on when the unsolicited report is complete.
- 10003 turns on when the unsolicited report is fails.
- The ladder logic program turns on register 1 when the connection is complete and turns off the register when the connection is broken.

```
/* -----------------------------------------------------------------
--------
   dnp.c
   Demonstration program for using the DNP connection handler.

   Copyright 2001, Control Microsystems Inc.
   --------------------------------------------------------------
-------- */

/* -----------------------------------------------------------------
--------
   Include Files
   --------------------------------------------------------------
-------- */
#include <ctools.h>

/* -----------------------------------------------------------------
--------
   Constants
   --------------------------------------------------------------
-------- */
#define CONNECTION_REQUIRED  10001      /* register for signaling
connection required */
#define MESSAGE_COMPLETE     10002      /* register for signaling
unsolicited message is complete */
#define MESSAGE_FAILED       10003      /* register for signaling
unsolicited message failed */
#define CONNECTION_STATUS    1   /* connection status register */

/* -----------------------------------------------------------------
--------
   Private Functions
   --------------------------------------------------------------
-------- */

/* -----------------------------------------------------------------
--------
   sampleDNPHandler
```

```
   This function is the user defined DNP connection handler.  It
will be
   called by internal DNP routines when a connection is required,
when
   confirmation of a message is received, and when a communication
timeout
   occurs.

   The function takes a variable of type DNP_CONNECTION_EVENT as
an input.
   This input instructs the handler as to what functionality is
required.
   The valid choices are connection required
(DNP_CONNECTION_REQUIRED),
   message confirmation received (DNP_MESSAGE_COMPLETE), and
timeout occurred
   (DNP_MESSAGE_TIMEOUT).

   The function does not return any values.
   ---------------------------------------------------------------
-------- */
static void sampleDNPHandler(DNP_CONNECTION_EVENT event)
{
       /* Determine what connection event is required or just
occurred */
       switch(event)
       {
              case DNP_CONNECTION_REQUIRED:
                     /* indicate connection is needed and clear
other bits */
                     request_resource(IO_SYSTEM);
                     setdbase(MODBUS, CONNECTION_REQUIRED, 1);
                     setdbase(MODBUS, MESSAGE_COMPLETE, 0);
                     setdbase(MODBUS, MESSAGE_FAILED, 0);
                     release_resource(IO_SYSTEM);
                     break;

              case DNP_MESSAGE_COMPLETE:
                     /* indicate message sent and clear other bits
*/
                     request_resource(IO_SYSTEM);
                     setdbase(MODBUS, CONNECTION_REQUIRED, 0);
                     setdbase(MODBUS, MESSAGE_COMPLETE, 1);
                     setdbase(MODBUS, MESSAGE_FAILED, 0);
                     release_resource(IO_SYSTEM);
                     break;

              case DNP_MESSAGE_TIMEOUT:
                     /* indicate message failed and clear other
bits */
                     request_resource(IO_SYSTEM);
                     setdbase(MODBUS, CONNECTION_REQUIRED, 0);
                     setdbase(MODBUS, MESSAGE_COMPLETE, 0);
                     setdbase(MODBUS, MESSAGE_FAILED, 1);
                     release_resource(IO_SYSTEM);
                     break;
```

```
            default:
                  /* ignore invalid requests */
                  break;
      }
}

/* ------------------------------------------------------------------
--------
   Public Functions
   ------------------------------------------------------------------
-------- */

/* ------------------------------------------------------------------
--------
   main

   This function is the main task of a user application. It
monitors a
   register from the ladder logic application. When the register
value
   changes, the function signals DNP events.

   The function has no parameters.

   The function does not return.
   ------------------------------------------------------------------
-------- */
void main(void)
{
      int lastConnectionState;   /* last state of connection
register */
      int currentConnectionState;      /* current state of
connection register */

      /* install DNP connection handler */
      dnpInstallConnectionHandler(sampleDNPHandler);

      /* get the current connection state */
      lastConnectionState = dbase(MODBUS, CONNECTION_STATUS);

      /* loop forever */
      while (TRUE)
      {
            request_resource(IO_SYSTEM);

            /* get the current connection state */
            currentConnectionState = dbase(MODBUS,
CONNECTION_STATUS);

            /* if the state has changed */
            if (currentConnectionState != lastConnectionState)
            {
                  /* if the connection is active */
                  if (currentConnectionState)
                  {
```

```
                                        /* Inform DNP that a connection exists
*/
                                        dnpConnectionEvent(DNP_CONNECTED);

                                        /* clear the request flag */
                                        setdbase(MODBUS, CONNECTION_REQUIRED,
0);
                                }
                                else
                                {
                                        /* Inform DNP that the connection is
closed */
                                        dnpConnectionEvent(DNP_DISCONNECTED);

                                        /* clear the message flags */
                                        setdbase(MODBUS, MESSAGE_COMPLETE, 0);
                                        setdbase(MODBUS, MESSAGE_FAILED, 0);
                                }

                                /* save the new state */
                                lastConnectionState = currentConnectionState;
                        }

                        /* release the processor so other tasks can run */
                        release_resource(IO_SYSTEM);
                        release_processor();
                }
        }
```

dnpClearEventLog

Clear DNP Event Log

Syntax:

```
#include <ctools.h>
BOOLEAN dnpClearEventLog(void);
```

Description:

The **dnpClearEventLogs** function deletes all change events from the DNP change event buffers, for all point types.

Example:

See the example in the section **dnpSendUnsolicited**.

dnpConnectionEvent

Report a DNP connection event

Syntax

```
#include <ctools.h>
void dnpConnectionEvent(DNP_CONNECTION_EVENT event);
```

Description

dnpConnectionEvent is used to report a change in connection status to DNP. This function is only used if a custom DNP connection handler has been installed.

event is current connection status. The valid connection status settings are DNP_CONNECTED, and DNP_DISCONNECTED.

See Also

dnpInstallConnectionHandler

Example

See the dnpInstallConnectionHandler example.

dnpCreateRoutingTable

Create Routing Table

Syntax

#include <ctools.h>

BOOLEAN createRoutingTable (UINT16 size);

Description

This function destroys any existing DNP routing table, and allocates memory for a new routing table according to the ‘size’ parameter.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

The function returns TRUE if successful, FALSE otherwise.

Example

See the example in the section **dnpSendUnsolicited**.

dnpGenerateEventLog

Generate DNP Event Log

Syntax

```
#include <ctools.h>
BOOLEAN dnpGenerateEventLog(
        UINT16 pointType,
        UINT16 pointAddress
);
```

Description

The dnpGenerateEventLog function generates a change event for the DNP point specified by pointType and pointAddress.

pointType specifies the type of DNP point. Allowed values are:

BI_POINT	binary input
AI16_POINT	16 bit analog input
AI32_POINT	32 bit analog input
AISF_POINT	short float analog input
CI16_POINT	16 bit counter output
CI32_POINT	32 bit counter output

pointAddress specifies the DNP address of the point.

A change event is generated for the specified point (with the current time and current value), and stored in the DNP event buffer.

The format of the event will depend on the Event Reporting Method and Class of Event Object that have been configured for the point.

The function returns TRUE if the event was generated. It returns FALSE if the DNP point is invalid, or if the DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

Example

See the example in the section **dnpSendUnsolicited**.

dnpGetAI16Config

Get DNP 16-bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAI16Config(
       UINT16 point,
       dnpAnalogInput * pAnalogInput
       );
```

Description

This function reads the configuration of a DNP 16-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveAI16Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpGetAI32Config

Get DNP 32-bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAI32Config(
       UINT32 point,
       dnpAnalogInput * pAnalogInput
       );
```

Description

This function reads the configuration of a DNP 32-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveAI32Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpGetAISFConfig

Get Short Floating Point Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAISFConfig (
       UINT16 point,
       dnpAnalogInput *pAnalogInput;
);
```

Description

This function reads the configuration of a DNP short floating point analog input point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully read, or FALSE otherwise (if the point number is not valid, or pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

dnpGetAO16Config

Get DNP 16-bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAO16Config(
       UINT16 point,
       dnpAnalogOutput * pAnalogOutput
       );
```

Description

This function reads the configuration of a DNP 16-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveAO16Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpGetAO32Config

Get DNP 32-bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAO32Config(
        UINT32 point,
        dnpAnalogOutput * pAnalogOutput
        );
```

Description

This function reads the configuration of a DNP 32-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveAO32Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpGetAOSFConfig

Get Short Floating Point Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetAOSFConfig (
       UINT16 point,
       dnpAnalogOutput *pAnalogOutput;
);
```

Description

This function reads the configuration of a DNP short floating point analog output point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully read, or FALSE otherwise (if the point number is not valid, or pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

dnpGetBIConfig

Get DNP Binary Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetBIConfig(
       UINT16 point,
       dnpBinaryInput * pBinaryInput
       );
```

Description

This function reads the configuration of a DNP binary input point.

The function has two parameters: the point number; and a pointer to a binary input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveBIConfig

Example

See example in the ***dnpGetConfiguration*** function section.

dnpGetBIConfigEx

Read DNP Binary Input Extended Point

Syntax

```
BOOLEAN dnpGetBIConfigEx(
        UINT16 point,
        dnpBinaryInputEx *pBinaryInput
);
```

Description

This function reads the configuration of an extended DNP Binary Input point.

The function has two parameters: the point number, and a pointer to an extended binary input point configuration structure.

The function returns TRUE if the configuration was successfully read. It returns FALSE if the point number is not valid, if the configuration is not valid, or if the DNP configuration has not been created.

This function supersedes dnpSaveBIConfig.

dnpGetBOConfig

Get DNP Binary Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetBOConfig(
       UINT16 point,
       dnpBinaryOutput * pBinaryOutput
       );
```

Description

This function reads the configuration of a DNP binary output point.

The function has two parameters: the point number; and a pointer to a binary output point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

Example

See example in the *dnpGetConfiguration* function section.

dnpGetCI16Config

Get DNP 16-bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetCI16Config(
       UINT16 point,
       dnpCounterInput * pCounterInput
       );
```

Description

This function reads the configuration of a DNP 16-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveCI16Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpGetCI32Config

Get DNP 32-bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetCI32Config(
       UINT32 point,
       dnpCounterInput * pCounterInput
       );
```

Description

This function reads the configuration of a DNP 32-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was read. It returns FALSE if the point number is not valid, if the pointer is NULL, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpSaveCI32Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpGetConfiguration

Get DNP Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpGetConfiguration(
       dnpConfiguration * pConfiguration
       );
```

Description

This function reads the DNP configuration.

The function has one parameter: a pointer to a DNP configuration structure.

The function returns TRUE if the configuration was read and FALSE if an error occurred.

See Also

dnpSaveConfiguration

Example

The following program demonstrates how to configure DNP for operation on com2. To illustrate creation of points it uses a sequential mapping of Modbus registers to points. This is not required. Any mapping may be used.

```
void main(void)
{
       UINT16 index;                     /* loop index */
       struct prot_settings settings;    /* protocol settings */
       dnpConfiguration configuration;   /* configuration settings
*/
       dnpBinaryInput binaryInput;             /* binary input
settings */
       dnpBinaryOutput binaryOutput;           /* binary output
settings */
       dnpAnalogInput analogInput;             /* analog input
settings */
       dnpAnalogOutput analogOutput;           /* analog output
settings */
       dnpCounterInput counterInput;           /* counter input
settings */

       /* Stop any protocol currently active on com port 2 */
       get_protocol(com2,&settings);
       settings.type = NO_PROTOCOL;
       set_protocol(com2,&settings);

       /* Load the Configuration Parameters */
       configuration.masterAddress         = DEFAULT_DNP_MASTER;
       configuration.rtuAddress            = DEFAULT_DNP_RTU;
       configuration.datalinkConfirm       = TRUE;
```

```
        configuration.datalinkRetries        =
DEFAULT_DLINK_RETRIES;
        configuration.datalinkTimeout        =
DEFAULT_DLINK_TIMEOUT;

        configuration.operateTimeout         =
DEFAULT_OPERATE_TIMEOUT;
        configuration.applicationConfirm     = TRUE;
        configuration.maximumResponse        =
DEFAULT_MAX_RESP_LENGTH;
        configuration.applicationRetries     = DEFAULT_APPL_RETRIES;
        configuration.applicationTimeout     = DEFAULT_APPL_TIMEOUT;
        configuration.timeSynchronization    = TIME_SYNC;

        configuration.BI_number              = 8;
        configuration.BI_cosBufferSize       = DEFAULT_COS_BUFF;
        configuration.BI_soeBufferSize       = DEFAULT_SOE_BUFF;
        configuration.BO_number              = 8;
        configuration.CI16_number            = 24;
        configuration.CI16_bufferSize        = 48;
        configuration.CI32_number            = 12;
        configuration.CI32_bufferSize        = 24;
        configuration.AI16_number            = 24;
        configuration.AI16_reportingMethod   = CURRENT_VALUE;
        configuration.AI16_bufferSize        = 24;
        configuration.AI32_number            = 12;
        configuration.AI32_reportingMethod   = CURRENT_VALUE;
        configuration.AI32_bufferSize        = 12;
        configuration.AO16_number            = 8;
        configuration.AO32_number            = 8;

        configuration.unsolicited            = TRUE;

        configuration.holdTime               = DEFAULT_HOLD_TIME;
        configuration.holdCount              = DEFAULT_HOLD_COUNT;

        dnpSaveConfiguration(&configuration);

        /* Start DNP protocol on com port 2 */
        get_protocol(com2,&settings);
        settings.type = DNP;
        set_protocol(com2,&settings);

        /* Save port settings so DNP protocol will automatically
start */
        request_resource(IO_SYSTEM);
        save(EEPROM_RUN);
        release_resource(IO_SYSTEM);

        /* Configure Binary Output Points */
        for (index = 0; index < configuration.BO_number; index++)
        {
                binaryOutput.modbusAddress1 = 1 + index;
                binaryOutput.modbusAddress2 = 1 + index;
                binaryOutput.controlType    = NOT_PAIRED;
```

```
        dnpSaveBOConfig(index, &binaryOutput);
}

/* Configure Binary Input Points */
for (index = 0;index < configuration.BI_number; index++)
{
        binaryInput.modbusAddress = 10001 + index;
        binaryInput.class         = CLASS_1;
        binaryInput.eventType     = COS;

        dnpSaveBIConfig(index, &binaryInput);
}

/* Configure 16 Bit Analog Input Points */
for (index = 0; index < configuration.AI16_number; index++)
{
        analogInput.modbusAddress  = 30001 + index;
        analogInput.class          = CLASS_2;
        analogInput.deadband       = 1;

        dnpSaveAI16Config(index, &analogInput);
}

/* Configure32 Bit Analog Input Points */
for (index = 0; index < configuration.AI32_number; index++)
{
        analogInput.modbusAddress  = 30001 + index * 2;
        analogInput.class          = CLASS_2;
        analogInput.deadband       = 1;

        dnpSaveAI32Config(index,&analogInput);
}

/* Configure 16 Bit Analog Output Points */
for (index = 0;index < configuration.AO16_number; index++)
{
        analogOutput.modbusAddress = 40001 + index;

        dnpSaveAO16Config(index, &analogOutput);
}

/* Configure 32 Bit Analog Output Points */
for (index = 0; index < configuration.AO32_number; index++)
{
        analogOutput.modbusAddress = 40101 + index * 2;

        dnpSaveAO32Config(index, &analogOutput);
}

/* Configure 16 Bit Counter Input Points */
for (index = 0; index < configuration.CI16_number; index++)
{
        counterInput.modbusAddress = 30001 + index;
        counterInput.class         = CLASS_3;
        counterInput.threshold     = 1;
```

```
            dnpSaveCI16Config(index, &counterInput);
        }

        /* Configure 32 bit Counter Input Points */
        for (index = 0; index < configuration.CI32_number; index++)
        {
            counterInput.modbusAddress = 30001 + index * 2;
            counterInput.class         = CLASS_3;
            counterInput.threshold     = 1;

            dnpSaveCI32Config(index, &counterInput);
        }

        /* add additional initialization code for your application
here ... */

        /* loop forever */
        while (TRUE)
        {
            /* add additional code for your application here ...
*/

            /* allow other tasks of this priority to execute */
            release_processor();
        }
        return;
}
```

dnpGetConfigurationEx

Read DNP Extended Configuration

Syntax

```
BOOLEAN dnpGetConfigurationEx (
      dnpConfigurationEx *pDnpConfigurationEx
);
```

Description

This function reads the extended DNP configuration parameters.

The function has one parameter: a pointer to the DNP extended configuration structure.

The function returns TRUE if the configuration was successfully read, or FALSE otherwise (if the pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

This function supersedes the dnpGetConfiguration function.

dnpGetRuntimeStatus

Get DNP Runtime Status

Syntax:

```
#include <ctools.h>
BOOLEAN dnpGetRuntimeStatus(
       DNP_RUNTIME_STATUS *status
);
```

Description:

The **dnpGetRuntimeStatus** function reads the current status of all DNP change event buffers, and returns information in the status structure.

DNP needs to be enabled before calling this function in order to create the DNP configuration.

Example:

See the example in the section **dnpSendUnsolicited**.

dnpGetUnsolicitedBackoffTime

Get DNP Unsolicited Back Off Time

Syntax:

```
#include <ctools.h>
UINT16 dnpGetUnsolicitedBackoffTime();
```

Description:

The dnpGetUnsolicitedBackoffTime function reads the unsolicited back off time from the controller.

The time is in seconds; and the allowed range is 0-65535 seconds. A value of zero indicates that the unsolicited back off timer is disabled.

dnpReadRoutingTableDialStrings

Read DNP Routing Table Entry Dial Strings

Syntax

```
BOOLEAN dnpReadRoutingTableDialStrings(
       UINT16 index,
       UINT16 maxPrimaryDialStringLength,
       CHAR *primaryDialString,
       UINT16 maxSecondaryDialStringLength,
       CHAR *secondaryDialString
);
```

Description

This function reads a primary and secondary dial string from an entry in the DNP routing table.

index specifies the index of an entry in the DNP routing table.

maxPrimaryDialStringLength specifies the maximum length of primaryDialString excluding the null-terminator character. The function uses this to limit the size of the returned string to prevent overflowing the storage passed to the function.

primaryDialString returns the primary dial string of the target station. It needs to point to an array of size maxPrimaryDialStringLength.

maxSecondaryDialStringLength specifies the maximum length of secondaryDialString excluding the null-terminator character. The function uses this to limit the size of the returned string to prevent overflowing the storage passed to the function.

secondaryDialString returns the secondary dial string of the target station. It needs to point to an array of size maxSecondaryDialStringLength.

Notes

This function needs to be used in conjunction with the dnpReadRoutingTableEntry function to read a complete entry in the DNP routing table.

dnpReadRoutingTableEntry

Read Routing Table entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpReadRoutingTableEntry (
       UINT16 index,
              routingTable *pRoute
);
```

Description

This function reads an entry from the routing table.

pRoute is a pointer to a table entry; it is written by this function.

The return value is TRUE if pRoute was successfully written or FALSE otherwise.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

The function returns the total number of entries in the DNP routing table.

dnpReadRoutingTableSize

Read Routing Table size

Syntax

```
#include <ctools.h>
UINT16 dnpReadRoutingTableSize (void);
```

Description

This function reads the total number of entries in the routing table.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

The function returns the total number of entries in the routing table.

dnpSaveAI16Config

Save DNP 16-Bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAI16Config(
       UINT16 point,
       dnpAnalogInput * pAnalogInput
       );
```

Description

This function sets the configuration of a DNP 16-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetAI16Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveAI32Config

Save DNP 32-Bit Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAI32Config(
        UINT32 point,
        dnpAnalogInput * pAnalogInput
        );
```

Description

This function sets the configuration of a DNP 32-bit analog input point.

The function has two parameters: the point number; and a pointer to an analog input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetAI32Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveAISFConfig

Save Short Floating Point Analog Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAISFConfig (
       UINT16 point,
       dnpAnalogInput *pAnalogInput;
);
```

Description

This function sets the configuration of a DNP short floating point analog input point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully written, or FALSE otherwise (if the point number is not valid, or the configuration is not valid, or if the DNP configuration has not been created).

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

dnpSaveAO16Config

Save DNP 16-Bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAO16Config(
       UINT16 point,
       dnpAnalogOutput * pAnalogOutput
       );
```

Description

This function sets the configuration of a DNP 16-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveAO32Config

Save DNP 32-Bit Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAO32Config(
        UINT32 point,
        dnpAnalogOutput * pAnalogOutput
        );
```

Description

This function sets the configuration of a DNP 32-bit analog output point.

The function has two parameters: the point number; and a pointer to an analog output point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetAO32Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveAOSFConfig

Save Short Floating Point Analog Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveAOSFConfig (
       UINT16 point,
       dnpAnalogOutput *pAnalogOutput;
);
```

Description

This function sets the configuration of a DNP short floating point analog output point.

The function has two parameters: the point number, and a pointer to a configuration structure.

The function returns TRUE if the configuration was successfully written, or FALSE otherwise (if the point number is not valid, or the configuration is not valid, or if the DNP configuration has not been created).

Notes

DNP needs to be enabled before calling this function in order to create the DNP

dnpSaveBIConfig

Save DNP Binary Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveBIConfig(
       UINT16 point,
       dnpBinaryInput * pBinaryInput
       );
```

Description

This function sets the configuration of a DNP binary input point.

The function has two parameters: the point number; and a pointer to a binary input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveBIConfigEx

Write DNP Binary Input Extended Point

Syntax

```
BOOLEAN dnpSaveBIConfigEx(
       UINT16 point,
       dnpBinaryInputEx *pBinaryInput
);
```

Description

This function writes the configuration of an extended DNP Binary Input point.

The function has two parameters: the point number, and a pointer to an extended binary input point configuration structure.

The function returns TRUE if the configuration was successfully written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if the DNP configuration has not been created.

This function supersedes dnpSaveBIConfig.

dnpSaveBOConfig

Save DNP Binary Output Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveBOConfig(
       UINT16 point,
       dnpBinaryOutput * pBinaryOutput
       );
```

Description

This function sets the configuration of a DNP binary output point.

The function has two parameters: the point number; and a pointer to a binary output point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveCI16Config

Save DNP 16-Bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveCI16Config(
       UINT16 point,
       dnpCounterInput * pCounterInput
       );
```

Description

This function sets the configuration of a DNP 16-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetCI16Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveCI32Config

Save DNP 32-Bit Counter Input Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveCI32Config(
        UINT32 point,
        dnpCounterInput * pCounterInput
        );
```

Description

This function sets the configuration of a DNP 32-bit counter input point.

The function has two parameters: the point number; and a pointer to a counter input point configuration structure.

The function returns TRUE if the configuration was written. It returns FALSE if the point number is not valid, if the configuration is not valid, or if DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

See Also

dnpGetCI32Config

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveConfiguration

Save DNP Configuration

Syntax

```
#include <ctools.h>
BOOLEAN dnpSaveConfiguration(
       dnpConfiguration * pConfiguration
       );
```

Description

This function sets the DNP configuration.

The function has one parameter: a pointer to a DNP configuration structure.

The function returns TRUE if the configuration was updated and FALSE if an error occurred. No changes are made to any parameters if an error occurs.

Notes

This function needs to be called before enabling DNP.

The following parameters cannot be changed if DNP is enabled. The function will not make any changes and will return FALSE if this is attempted. The protocol needs to be disabled in order to make a change involving these parameters.

- BI_number
- BI_cosBufferSize
- BI_soeBufferSize
- BO_number
- CI16_number
- CI16_bufferSize
- CI32_number
- CI32_bufferSize
- AI16_number
- AI16_reportingMethod
- AI16_bufferSize
- AI32_number
- AI32_reportingMethod
- AI32_bufferSize
- AO16_number
- AO32_number

The following parameters can be changed when DNP is enabled.

- masterAddress;
- rtuAddress;
- datalinkConfirm;
- datalinkRetries;
- datalinkTimeout;
- operateTimeout
- applicationConfirm
- maximumResponse
- applicationRetries
- applicationTimeout
- timeSynchronization
- unsolicited
- holdTime
- holdCount

See Also

dnpGetConfiguration

Example

See example in the ***dnpGetConfiguration*** function section.

dnpSaveConfigurationEx

Write DNP Extended Configuration

Syntax

```
BOOLEAN dnpSaveConfigurationEx (
      dnpConfigurationEx *pDnpConfigurationEx
);
```

Description

This function writes the extended DNP configuration parameters.

The function has one parameter: a pointer to the DNP extended configuration structure.

The function returns TRUE if the configuration was successfully written, or FALSE otherwise (if the pointer is NULL, or if the DNP configuration has not been created).

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

This function supersedes the dnpSaveConfiguration function.

dnpSaveUnsolicitedBackoffTime

Save DNP Unsolicited Back Off Time

Syntax:

```
BOOLEAN dnpSaveUnsolicitedBackoffTime (
      UINT16 backoffTime
);
```

Description:

The dnpSaveUnsolicitedBackoffTime function writes the unsolicited back off time to the controller.

The time is in seconds; and the allowed range is 0-65535 seconds. A value of zero indicates that the unsolicited back off timer is disabled.

The function returns TRUE if the function was successful. It returns FALSE if the DNP configuration has not been created.

dnpSearchRoutingTable

Search Routing Table

Syntax

```
#include <ctools.h>
BOOLEAN dnpSearchRoutingTable (
       UINT16 Address
              routingTable  *pRoute
);
```

Description

This function searches the routing table for a specific DNP address.

pRoute is a pointer to a table entry; it is written by this function.

The return value is TRUE if pRoute was successfully written or FALSE otherwise.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

dnpSendUnsolicited

Send DNP Unsolicited Response

Syntax

```
#include <ctools.h>
UINT16 dnpSendUnsolicitedResponse(
       UINT16 classFlags
);
```

Description

The **dnpSendUnsolicitedResponse** function sends an ‘Unsolicited Response’ message in DNP protocol, with data from the specified class(es).

- *class* specifies the class(es) of event data to include in the message.
- Allowed values are:

```
#define CLASS0_FLAG 0x01   /* flag for enabling Class 0
Unsolicited Responses */
#define CLASS1_FLAG 0x02   /* flag for enabling Class 1
Unsolicited Responses */
#define CLASS2_FLAG 0x04   /* flag for enabling Class 2
Unsolicited Responses */
#define CLASS3_FLAG 0x08   /* flag for enabling Class 3
Unsolicited Responses */
```

DNP needs to be enabled before calling this function in order to create the DNP configuration.

Example

```
/* -----------------------------------------------------------------
   SCADAPack 32 C++ Application Main Program
   Copyright 2001 - 2002, Control Microsystems Inc.

   Test application for new DNP API Functions.
   written by James Wiles May 2003

   This app was written for a ScadaPack 32P, running DNP on comm
port
   4.

   ------------------------------------------------------------ */
#include <ctools.h>
#include <string.h>


/* -----------------------------------------------------------------
---
   Constants
   -----------------------------------------------------------------
*/
```

```
/*
 * Event Triggers :
 * This application detects when these registers have been set,
 * then performs the specified action and clears the register.
 */
#define CLEAR_EVENTS             100    /* Clear all DNP Event Log
Buffers */
#define GENERATE_BI_EVENT  101    /* Generate a change event for BI
channel 0 */
#define GENERATE_AI16_EVENT      102    /* Generate a change event
for 16-bit AI channel 0 */
#define CLASS0_REPORT            103    /* Send an unsolicited
report of Class 0 data */

/*
 * Status Flags
 */
#define EVENTS_CLASS1            110
#define EVENTS_CLASS2            111
#define EVENTS_CLASS3            112

/*
 * Status Registers
 */
#define EVENT_COUNT_AI16   40102
#define EVENT_COUNT_BI           40104
#define EVENT_COUNT_CLASS1 40106
#define EVENT_COUNT_CLASS2 40108
#define EVENT_COUNT_CLASS3 40110

/* ---------------------------------------------------------------
---
   main

   This routine is the main application loop.
   ---------------------------------------------------------------
*/
void main(void)
{
      UINT16 index;                            /* loop index */
      struct prot_settings protocolSettings;  /* protocol
settings */
      dnpConfiguration configuration;
      dnpBinaryInput binaryInput;
      dnpAnalogInput analogInput;
      DNP_RUNTIME_STATUS dnpStatus;
      int clear_events_flag;
      int bi_event_flag;
      int ai16_event_flag;
      int class0_report_flag;

      /* Set DNP Configuration */
      configuration.masterAddress         = 100;
      configuration.rtuAddress            = 1;
      configuration.datalinkConfirm       = FALSE;
```

```
       configuration.datalinkRetries        =
DEFAULT_DLINK_RETRIES;
       configuration.datalinkTimeout        =
DEFAULT_DLINK_TIMEOUT;

       configuration.operateTimeout         =
DEFAULT_OPERATE_TIMEOUT;
       configuration.applicationConfirm     = FALSE;
       configuration.maximumResponse        =
DEFAULT_MAX_RESP_LENGTH;
       configuration.applicationRetries     = DEFAULT_APPL_RETRIES;
       configuration.applicationTimeout     = DEFAULT_APPL_TIMEOUT;
       configuration.timeSynchronization    = NO_TIME_SYNC;

       configuration.BI_number              = 2;
       configuration.BI_startAddress        = 0;
       configuration.BI_reportingMethod     = REPORT_ALL_EVENTS;
       configuration.BI_soeBufferSize       = 1000;
       configuration.BO_number              = 0;
       configuration.BO_startAddress        = 0;
       configuration.CI16_number            = 0;
       configuration.CI16_startAddress      = 0;
       configuration.CI16_reportingMethod   = REPORT_ALL_EVENTS;
       configuration.CI16_bufferSize        = 0;
       configuration.CI32_number            = 0;
       configuration.CI32_startAddress      = 100;
       configuration.CI32_reportingMethod   = REPORT_ALL_EVENTS;
       configuration.CI32_bufferSize        = 0;
       configuration.CI32_wordOrder         = MSW_FIRST;
       configuration.AI16_number            = 2;
       configuration.AI16_startAddress      = 0;
       configuration.AI16_reportingMethod   = REPORT_ALL_EVENTS;
       configuration.AI16_bufferSize        = 1000;
       configuration.AI32_number            = 0;
       configuration.AI32_startAddress      = 100;
       configuration.AI32_reportingMethod   = REPORT_ALL_EVENTS;
       configuration.AI32_bufferSize        = 0;
       configuration.AI32_wordOrder         = MSW_FIRST;
       configuration.AISF_number            = 0;
       configuration.AISF_startAddress      = 200;
       configuration.AISF_reportingMethod   = REPORT_CHANGE_EVENTS;
       configuration.AISF_bufferSize        = 0;
       configuration.AISF_wordOrder         = MSW_FIRST;
       configuration.AO16_number            = 0;
       configuration.AO16_startAddress      = 0;
       configuration.AO32_number            = 0;
       configuration.AO32_startAddress      = 100;
       configuration.AO32_wordOrder         = MSW_FIRST;
       configuration.AOSF_number            = 0;
       configuration.AOSF_startAddress      = 200;
       configuration.AOSF_wordOrder         = MSW_FIRST;

       configuration.autoUnsolicitedClass1 = TRUE;
       configuration.holdTimeClass1         = 10;
       configuration.holdCountClass1        = 3;
       configuration.autoUnsolicitedClass2 = TRUE;
```

```
        configuration.holdTimeClass2        = 10;
        configuration.holdCountClass2       = 3;
        configuration.autoUnsolicitedClass3 = TRUE;
        configuration.holdTimeClass3        = 10;
        configuration.holdCountClass3       = 3;

        dnpSaveConfiguration(&configuration);

        /* Start DNP protocol on com port 4 */
        get_protocol(com4, &protocolSettings);
        protocolSettings.type = DNP;
        set_protocol(com4, &protocolSettings);


        /* Configure Binary Input Points */
        for (index = 0;index < configuration.BI_number; index++)
        {
              binaryInput.modbusAddress = 10001 + index;
              binaryInput.eventClass    = CLASS_1;
              dnpSaveBIConfig(configuration.BI_startAddress +
index, &binaryInput);
        }

        /* Configure 16 Bit Analog Input Points */
        for (index = 0; index < configuration.AI16_number; index++)
        {
              analogInput.modbusAddress  = 40002 + index * 2;
              analogInput.eventClass     = CLASS_2;
              analogInput.deadband       = 1;
              dnpSaveAI16Config(configuration.AI16_startAddress +
index, &analogInput);
        }

        /*
         * Configure DNP Routing Table :
         * station 100 via com4
         * station 101 via com4
         */
        dnpCreateRoutingTable(2);
        dnpWriteRoutingTableEntry(0, 100, CIF_Com4,
DEFAULT_DLINK_RETRIES, DEFAULT_DLINK_TIMEOUT);
        dnpWriteRoutingTableEntry(1, 101, CIF_Com4,
DEFAULT_DLINK_RETRIES, DEFAULT_DLINK_TIMEOUT);


        /*
         * main loop
         */
        while (TRUE)
        {
              /* request IO resource */
              request_resource(IO_SYSTEM);

              /* read DNP status */
              dnpGetRuntimeStatus(&dnpStatus);
```

```
            setdbase(MODBUS, EVENTS_CLASS1,
dnpStatus.eventCountClass1 ? 1 : 0);
            setdbase(MODBUS, EVENTS_CLASS2,
dnpStatus.eventCountClass2 ? 1 : 0);
            setdbase(MODBUS, EVENTS_CLASS3,
dnpStatus.eventCountClass3 ? 1 : 0);
            setdbase(MODBUS, EVENT_COUNT_AI16,
dnpStatus.eventCountAI16);
            setdbase(MODBUS, EVENT_COUNT_BI,
dnpStatus.eventCountBI);
            setdbase(MODBUS, EVENT_COUNT_CLASS1,
dnpStatus.eventCountClass1);
            setdbase(MODBUS, EVENT_COUNT_CLASS2,
dnpStatus.eventCountClass2);
            setdbase(MODBUS, EVENT_COUNT_CLASS3,
dnpStatus.eventCountClass3);
            release_resource(IO_SYSTEM);

            clear_events_flag = FALSE;
            bi_event_flag = FALSE;
            ai16_event_flag = FALSE;
            class0_report_flag = FALSE;

            /* Read Event Triggers */
            if (dbase(MODBUS, CLEAR_EVENTS))
            {
                  setdbase(MODBUS, CLEAR_EVENTS, 0);
                  clear_events_flag = TRUE;
            }

            if (dbase(MODBUS, GENERATE_BI_EVENT))
            {
                  setdbase(MODBUS, GENERATE_BI_EVENT, 0);
                  bi_event_flag = FALSE;
            }

            if (dbase(MODBUS, GENERATE_AI16_EVENT))
            {
                  setdbase(MODBUS, GENERATE_AI16_EVENT, 0);
                  ai16_event_flag = FALSE;
            }

            if (dbase(MODBUS, CLASS0_REPORT))
            {
                  setdbase(MODBUS, CLASS0_REPORT, 0);
                  class0_report_flag = FALSE;
            }

            /* release IO resource */
            release_resource(IO_SYSTEM);


            /* Clear DNP Event Log buffer if requested */
            if (clear_events_flag)
            {
                  dnpClearEventLog();
```

```
            }

            /* Generate a DNP Change Event for BI Point 0 if 
requested */
            if (bi_event_flag)
            {
                  dnpGenerateEventLog(BI_POINT, 0);
            }

            /* Generate a DNP Change Event for 16-bit AI Point 0 
if requested */
            if (ai16_event_flag)
            {
                  dnpGenerateEventLog(AI16_POINT, 0);
            }

            /* Send DNP Class 0 Unsolicited Report if requested 
*/
            if (class0_report_flag)
            {
                  dnpSendUnsolicitedResponse(CLASS0_FLAG);
            }

            /* release processor to other tasks */
            release_processor();
      }
}
```

dnpSendUnsolicitedResponse

Send DNP Unsolicited Response

Syntax

```
BOOLEAN dnpSendUnsolicitedResponse(
      UINT16 classFlags
);
```

Description

The dnpSendUnsolicitedResponse function sends an Unsolicited Response message in DNP, with data from the specified classes.

class specifies the class or classes of event data to include in the message. It can contain any combination of the following values; if multiple values are used they should be ORed together:

CLASS0_FLAG	enables Class 0 Unsolicited Responses
CLASS1_FLAG	enables Class 1 Unsolicited Responses
CLASS2_FLAG	enables Class 2 Unsolicited Responses
CLASS3_FLAG	enables Class 3 Unsolicited Responses

The function returns TRUE if the DNP unsolicited response message was successfully triggered. It returns FALSE if an unsolicited message of the same class is already pending, or if the DNP configuration has not been created.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

If no events are pending an empty unsolicited message will be sent.

dnpWriteRoutingTableEntry

Write Routing Table Entry

Syntax

```
#include <ctools.h>
BOOLEAN dnpWriteRoutingTableEntry (
       UINT16 index,
       UINT16 dnpAddress,
       UINT16 commPort,
       UINT16 DataLinkRetries,
       UINT16 DataLinkTimeout
);
```

Description

This function writes an entry in the DNP routing table.

Notes

DNP needs to be enabled before calling this function in order to create the DNP configuration.

The function returns TRUE if successful, FALSE otherwise.

Example

See the example in the section **dnpSendUnsolicited**.

dnpWriteRoutingTableDialStrings

Write DNP Routing Table Entry Dial Strings

Syntax

```
BOOLEAN dnpWriteRoutingTableDialStrings(
       UINT16 index,
       UINT16 primaryDialStringLength,
       CHAR *primaryDialString,
       UINT16 secondaryDialStringLength,
       CHAR *secondaryDialString
);
```

Description

This function writes a primary and secondary dial string into an entry in the DNP routing table.

index specifies the index of an entry in the DNP routing table.

primaryDialStringLength specifies the length of primaryDialString excluding the null-terminator character.

primaryDialString specifies the dial string used when dialing the target station. This string is used on the first attempt.

secondaryDialStringLength specifies the length of secondaryDialString excluding the null-terminator character.

secondaryDialString specifies the dial string to be used when dialing the target station. It is used for the next attempt if the first attempt is unsuccessful fails.

Notes

This function needs to be used in conjunction with the dnpWriteRoutingTableEntry function to write a complete entry in the DNP routing table.

dout

Write Digital Outputs

Syntax

```
#include <ctools.h>
int dout(unsigned channel, unsigned value);
```

Description

The **dout** function outputs *value* to the digital input or output specified by *channel*. It sets the status of 8 digital points.

The **dout** function returns the value output to the channel, as modified by the channel configuration tables. If channel is not valid, –1 is returned.

Notes

The **dout** function modifies all 8 bits (points) in a channel. Use the **turnon** and **turnoff** functions to write to single bits.

Use offsets from the symbolic constants DIN_START, DIN_END, EXTENDED_DIN_START, EXTENDED_DIN_END, DOUT_START, DOUT_END, EXTENDED_DOUT_START and EXTENDED_DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioWrite8Dout** directly.

See Also

ioWrite8Dout, din, pulse, timeout, turnon, turnoff, on, off

Example

This program sends all bit combinations to the second digital output channel.

```
#include <ctools.h>
void main(void)
{
	unsigned value;			/* output values */
	for (value = 0; value; value++)
	{
		request_resource(IO_SYSTEM);
		dout(DOUT_START + 1, value);
		release_resource(IO_SYSTEM);
	}
}
```

end_application

Terminates all Application Tasks

Syntax

```
#include <ctools.h>
void end_application(void);
```

Description

The **end_application** function terminates all **APPLICATION** type tasks created with the **create_task** function. Stack space and resources used by the tasks are freed.

Notes

This function is used normally by communication protocols to stop an executing application program, prior to loading a new program into memory.

See Also

create_task, end_task

end_task

Terminate a Task

Syntax

```
#include <ctools.h>
void end_task(unsigned task_ID);
```

Description

The **end_task** function terminates the task specified by *task_ID*. Stack space and resources used by the task are freed. The **end_task** function terminates both **APPLICATION** and **SYSTEM** type tasks.

See Also

create_task, end_application, getTaskInfo

endTimedEvent

Terminate Signaling of a Regular Event

Syntax

```
#include <ctools.h>
unsigned endTimedEvent(unsigned event);
```

Description

This **endTimedEvent** function cancels signaling of a timed event, initialized by the startTimedEvent function.

The function returns TRUE if the event signaling was canceled.

The function returns FALSE if the event number is not valid, or if the event was not previously initiated with the startTimedEvent function. The function has no effect in these cases.

Notes

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in ctools.h are not valid events for use in an application program.

Example

See the examples for **startTimedEvent**.

See Also

startTimedEvent

enronInstallCommandHandler

Installs handler for Enron Modbus commands.

Syntax

```
#include <ctools.h>
void enronInstallCommandHandler(
       UINT16 (* function)(
              UINT16   length,
              UCHAR  * pCommand,
              UINT16   responseSize,
              UINT16 * pResponseLength,
              UCHAR  * pResponse
              )
       );
```

Description

This function installs a handler function for Enron Modbus commands. The protocol driver calls this handler function each time a command is received for the Enron Modbus station.

function is a pointer to the handler function. If function is NULL the handler is disabled.

The function has no return value.

Notes

The application needs to disable the handler when the application ends. This prevents the protocol driver from calling the handler while the application is stopped. Call the enronInstallCommmandHandler with a NULL pointer. The usual method is to create a task exit handler function to do this. See the example below for details.

The handler function has five parameters.

- length is the number of characters in the command message.
- pCommand is a pointer to the command message. The first byte in the message is the function code, followed by the Enron Modbus message. See the Enron Modbus protocol specification for details on the message formats.
- responseSize is the size of the response buffer in characters.
- pResponseLength is a pointer to a variable that will hold the number of characters in the response. If the handler returns TRUE, it must set this variable.
- pResponse is a pointer to a buffer that will hold the response message. The buffer size is responseSize characters. The handler cannot write beyond the end of the buffer. If the handler returns TRUE, it needs to set this variable. The data needs to start with the function code and end with the last data

byte. The protocol driver will add the station address, checksum, and message framing to the response.

The handler function returns the following values.

Value	Description
NORMAL	Indicates protocol handler should send a normal response message. Data are returned using pResponse and pResponseLength.
ILLEGAL_FUNCTION	Indicates protocol handler should send an Illegal Function exception response message. This response should be used when the function code in the command is not recognised.
ILLEGAL_DATA_ADDRESS	Indicates protocol handler should send an Illegal Data Address exception response message. This response should be used when the data address in the command is not recognised.
ILLEGAL_DATA_VALUE	Indicates protocol handler should send an Illegal Data Value exception response message. This response should be used when invalid data is found in the command.

If the function returns NORMAL then the protocol driver sends the response message in the buffer pointed to by pResponse. If the function returns an exception response protocol driver returns the exception response to the caller. The buffer pointed to by pResponse is not used.

Example

```
This program installs a simple handler function.
#include <ctools.h>

/* ----------------------------------------------------
   This function processes Enron Modbus commands.
   ---------------------------------------------------- */
UINT16 commandHandler(
       UINT16   length,
       UCHAR  * pCommand,
       UINT16   responseSize,
       UINT16 * pResponseLength,
       UCHAR  * pResponse
       )
{
       UCHAR command;
       UINT16 result;

       /* if a command byte was received */
       if (length >= 1)
       {
```

```
                /* get the command byte */
                command = pCommand[0];
                switch (command)
                {
                /* read unit status command */
                case 7:
                        /* if the response buffer is large enough */
                        if (responseSize > 2)
                        {
                                /* build the response header */
                                pResponse[0] = pCommand[0];

                                /* set the unit status */
                                pResponse[1] = 17;

                                /* set response length */
                                *pResponseLength = 2;

                                /* indicate the command worked */
                                result = NORMAL;
                        }
                        else
                        {
                                /* buffer is to small to respond */
                                result = ILLEGAL_FUNCTION;
                        }
                        break;

                /* add cases for other commands here */

                default:
                        /* command is invalid */
                        result = ILLEGAL_FUNCTION;
                }
        }
        else
        {
                /* command is too short so return error */
                result = ILLEGAL_FUNCTION;
        }
        return result;
}

/* ------------------------------------------------------
   This function unhooks the protocol handler when the
   main task ends.
   ------------------------------------------------------ */
void mainExitHandler(void)
{
        /* unhook the handler function */
        enronInstallCommandHandler(NULL);
}

void main(void)
{
        TASKINFO thisTask;
```

```
        /* install handler to execute when this task ends */
        thisTask = getTaskInfo(0);
        installExitHandler(thisTask.taskID, mainExitHandler);

        /* install handler for Enron Modbus */
        enronInstallCommandHandler(commandHandler);

        /* infinite loop of main task */
        while (TRUE)
        {
                /* add application code here */
        }
}
```

forceLed

Set State of Force LED

Syntax

```
#include <ctools.h>
void forceLed(unsigned state);
```

Description

The **forceLed** function sets the state of the FORCE LED. *state* may be either LED_ON or LED_OFF.

Notes

The FORCE LED is used to indicate forced I/O.

See Also

setStatus

getABConfiguration

Get DF1 Protocol Configuration

Syntax

```
#include <ctools.h>
struct ABConfiguration *getABConfiguration(FILE *stream, struct
ABConfiguration *ABConfig);
```

Description

The **getABConfiguration** function gets the DF1 protocol configuration parameters for the *stream*. If *stream* does not point to a valid serial port the function has no effect. *ABConfig* needs to point to an DF1 protocol configuration structure.

The **getABConfiguration** function copies the DF1 configuration parameters into the *ABConfig* structure and returns a pointer to it.

Example

This program displays the DF1 configuration parameters for **com1**.

```
#include <ctools.h>

void main(void)
{
       struct ABConfiguration ABConfig;

       getABConfiguration(com1, &ABConfig);
       printf("Min protected address:     %u\r\n",
              ABConfig.min_protected_address);
       printf("Max protected address:     %u\r\n",
              ABConfig.max_protected_address);
}
```

getBootType

Get Controller Boot Up State

Syntax

```
#include <ctools.h>
unsigned getBootType(void);
```

Description

The **getBootType** function returns the boot up state of the controller. The possible return values are:

SERVICE	controller started in SERVICE mode
RUN	controller started in RUN mode

Example

```
#include <ctools.h>

void main(void)
{
	struct prot_settings settings;

	/* Disable the protocol on serial port 1 */
	settings.type = NO_PROTOCOL;
	settings.station = 1;
	settings.priority = 3;
	settings.SFMessaging = FALSE;
	request_resource(IO_SYSTEM);
	set_protocol(com1, &settings);
	release_resource(IO_SYSTEM);

	/* Display the boot status information */
	printf("Boot type: %d\r\n", getBootType());
}
```

getclock

Read the Real Time Clock

Syntax

```
#include <rtc.h>
struct clock getclock(void);
```

Description

The **getclock** function reads the time and date from the real time clock hardware.

The **getclock** function returns a struct clock containing the time and date information.

Notes

The time format returned by the **getclock** function is not compatible with the standard UNIX style functions supplied by Microtec.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

setclock, getClockTime

Example

This program displays the current date and time.

```
#include <ctools.h>
main(void)
{
	struct clock now;

	request_resource(IO_SYSTEM);
	now = getclock();                /* read the clock */
	release_resource(IO_SYSTEM);
	printf("%2d/%2d/%2d", now.day,
			now.month, now.year);
	printf("%2d:%2d\r\n",now.hour, now.minute);
}
```

getClockAlarm

Read the Real Time Clock Alarm Settings

Syntax

```
#include <ctools.h>
ALARM_SETTING getClockAlarm(void);
```

Description

The **getClockAlarm** function returns the alarm setting in the real time clock. The alarm is used to wake the controller from sleep mode.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

alarmIn, setClockAlarm

getClockTime

Read the Real Time Clock

Syntax

```
#include <ctools.h>
void getClockTime(long * pDays, long * pHundredths);
```

Description

The getClockTime function reads the read time clock and returns the value as the number of whole days since 01/01/97 and the number of hundredths of a second since the start of the current day. The function works for 100 years from 01/01/97 to 12/31/96 then rolls over.

The function has two parameters: a pointer to the variable to hold the days; and a pointer to a variable to hold the hundredths of a second.

The function has no return value.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

setclock, getclock

Get Controller ID

Syntax

```
#include <ctools.h>
void getControllerID(CHAR * pID)
```

Description

This function writes the Controller ID to the string pointed to by *pID*. The Controller ID is a unique ID for the controller set at the factory. The pointer *pID* needs to point to a character string of length CONTROLLER_ID_LEN.

Example

```
This program displays the Controller ID.
#include <ctools.h>

void main(void)
{
	char     ctlrID[CONTROLLER_ID_LEN];
	UINT16 index;

	getControllerID(ctlrID);

	fprintf(com1, "\r\nController ID : ");
	for (index=0; index<CONTROLLER_ID_LEN; index++)
	{
		fputc(ctlrID[index], com1);
	}
}
```

getForceFlag

Get Force Flag State for a Register

Syntax

```
#include <ctools.h>
unsigned getForceFlag(unsigned type, unsigned address, unsigned
*value);
```

Description

The **getForceFlag** function copies the value of the force flag for the specified database register into the integer pointed to by *value*. The valid range for *address* is determined by the database addressing *type*.

The force flag value is either 1 or 0, or a 16-bit mask for LINEAR digital addresses.

If the *address* or addressing *type* is not valid, FALSE is returned and the integer pointed to by *value* is 0; otherwise TRUE is returned. The table below shows the valid address types and ranges.

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

Force Flags are not modified when the controller is reset. Force Flags are in a permanent storage area, which is maintained during power outages.

Refer to the *I/O Database and Register Assignment* chapter for more information.

See Also

setForceFlag, clearAllForcing, overrideDbase

Example

This program obtains the force flag state for register 40001, for the 16 status registers at linear address 302 (i.e. registers 10737 to 10752), and for the holding register at linear address 1540 (i.e. register 40005).

```
#include <ctools.h>

void main(void)
{
	unsigned flag, bitmask;
```

```
        getForceFlag(MODBUS, 40001, &flag);
        getForceFlag(LINEAR, 302, &bitmask);
        getForceFlag(LINEAR, 1540, &flag);
}
```

getIOErrorIndication

Get I/O Module Error Indication

Syntax

```
#include <ctools.h>
unsigned getIOErrorIndication(void);
```

Description

The **getIOErrorIndication** function returns the state of the I/O module error indication. TRUE is returned if the I/O module communication status is currently reported in the controller status register and Status LED. FALSE is returned if the I/O module communication status is not reported.

Notes

Refer to the ***5203/4 System Manual*** or the ***SCADAPack System Manual*** for further information on the Status LED and Status Output.

See Also

setIOErrorIndication

getOutputsInStopMode

Get Outputs In Stop Mode

Syntax

```
#include <ctools.h>
void getOutputsInStopMode( unsigned *doutsInStopMode, unsigned
*aoutsInStopMode);
```

Description

The **getOutputsInStopMode** function copies the values of the output control flags into the integers pointed to by *doutsInStopMode* and *aoutsInStopMode*.

If the value pointed to by *doutsInStopMode* is TRUE, then digital outputs are held at their last state when the Ladder Logic program is stopped.

If the value pointed to by *doutsInStopMode* is FALSE, then digital outputs are turned OFF when the Ladder Logic program is stopped.

If the value pointed to by a*outsInStopMode* is TRUE, then analog outputs are held at their last value when the Ladder Logic program is stopped.

If the value pointed to by a*outsInStopMode* is FALSE, then analog outputs go to zero when the Ladder Logic program is stopped.

See Also

setOutputsInStopMode

Example

See the example for **setOutputsInStopMode** function.

getPortCharacteristics

Get Serial Port Characteristics

Syntax

```
#include <ctools.h>
unsigned getPortCharacteristics(FILE *stream, PORT_CHARACTERISTICS
*pCharacteristics);
```

Description

The **getPortCharacteristics** function gets information about features supported by the serial port pointed to by *stream*. If *stream* does not point to a valid serial port the function has no effect and FALSE is returned; otherwise TRUE is returned.

The **getPortCharacteristics** function copies the serial port characteristics into the structure pointed to by *pCharacteristics*.

Notes

Refer to the **Overview of Functions** section for detailed information on serial ports.

Refer to the **Structures and Types** section for a description of the fields in the PORT_CHARACTERISTICS structure.

See Also

get_port

Example

```
#include <ctools.h>
void main(void)
{
	PORT_CHARACTERISTICS options;

	getPortCharacteristics(com3, &options);
	fprintf(com1, "Dataflow options: %d\r\n",
		options.dataflow);
	fprintf(com1, "Protocol options: %d\r\n",
		options.protocol);
}
```

getPowerMode

Get Current Power Mode

Syntax

```
#include <ctools.h>
BOOLEAN getPowerMode(UCHAR* cpuPower, UCHAR* lan, UCHAR*
usbPeripheral, UCHAR* usbHost);
```

Description

The **getPowerMode** function places the current state of the CPU, LAN, USB peripheral port, and USB host port in the passed parameters. The following table lists the possible return values and their meaning.

Macro	Meaning
PM_CPU_FULL	The CPU is set to run at full speed
PM_CPU_REDUCED	The CPU is set to run at a reduced speed
PM_CPU_SLEEP	The CPU is set to sleep mode
PM_LAN_ENABLED	The LAN is enabled
PM_LAN_DISABLED	The LAN is disabled
PM_USB_PERIPHERAL_ENABLED	The USB peripheral port is enabled
PM_USB_PERIPHERAL_DISABLED	The USB peripheral port is disabled
PM_USB_HOST_ENABLED	The USB host port is enabled
PM_USB_HOST_DISABLED	The USB host port is disabled
PM_UNAVAILABLE	The status of the device could not be read.

TRUE is returned if the values placed in the passed parameters are valid, otherwise FALSE is returned.

The application program may set the current power mode with the s**etPowerMode** function.

See Also

setPowerMode, setWakeSource, getWakeSource

get_pid

Get PID Variable

Syntax

```
#include <ctools.h>
int get_pid(unsigned name, unsigned block);
```

Description

The **get_pid** function returns the value of a PID control block variable. *name* needs to be specified by one of the variable name macros in **pid.h**. *block* needs to be in the range 0 to **PID_BLOCKS-1**.

Notes

See the ***Telepace PID Controllers Manual*** for a detailed description of PID control.

Values stored in PID blocks are not initialized when a program is run, and are guaranteed to retain their values during power failures and program loading. The user program must always initialize PID block variables.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

set_pid, auto_pid, clear_pid

get_port

Get Serial Port Configuration

Syntax

```
#include <ctools.h>
struct pconfig *get_port(FILE *stream, struct pconfig *settings);
```

Description

The **get_port** function gets the serial port configuration for the *stream*. If *stream* does not point to a valid serial port the function has no effect.

The **get_port** function copies the serial port settings into the structure pointed to by *settings* and returns a pointer to the structure.

Notes

Refer to the **Overview of Functions** section for detailed information on serial ports.

Refer to the **Structure and Types** section for a description of the fields in the *pconfig* structure.

See Also

set_port

Example

```
#include <ctools.h>

void main(void)
{
	struct pconfig settings;

	get_port(com1, &settings);
	printf("Baud rate: %d\r\n", settings.baud);
	printf("Duplex:    %d\r\n", settings.duplex);
}
```

getProgramStatus

Get Program Status Flag

Syntax

#include <ctools.h>

unsigned getProgramStatus(void);

Description

The **getProgramStatus** function returns the application program status flag. The status flag is set to **NEW_PROGRAM** when the C program is erased or downloaded to the controller from the program loader.

The application program may modify the status flag with the **setProgramStatus** function.

Example

This program stores a default alarm limit into the I/O database the first time it is run. On subsequent executions, it uses the limit in the database. The limit in the database can be modified by a communication protocol during execution.

```
#include <ctools.h>

#define HI_ALARM          41000
#define ALARM_OUTPUT      1026

void main( void )
{
       int inputValue;

       if (getProgramStatus() == NEW_PROGRAM)
       {
              /* Set default alarm limit */
              request_resource(IO_SYSTEM);
              setdbase(MODBUS, HI_ALARM, 4000);
              release_resource(IO_SYSTEM);

              /* Use values in database from now on */
              setProgramStatus(PROGRAM_EXECUTED);
       }
       while (TRUE)
       {
              request_resource(IO_SYSTEM);

              /* Test input against alarm limits */
              if (ain(INPUT) > dbase(MODBUS, HI_ALARM))
                     setdbase(MODBUS, ALARM_OUTPUT, 1);
              else
                     setdbase(MODBUS, ALARM_OUTPUT, 0);

              release_resource(IO_SYSTEM);
```

```
                    /* Allow other tasks to execute */
                    release_processor();
            }
    }
```

get_protocol

Get Protocol Configuration

Syntax

```
#include <ctools.h>
struct prot_settings *get_protocol(FILE *stream, struct
prot_settings *settings);
```

Description

The **get_protocol** function gets the communication protocol configuration for the *stream*. If *stream* does not point to a valid serial port the function has no effect. *settings* needs to point to a protocol configuration structure, *prot_settings*.

The **get_protocol** function copies the protocol settings into the structure pointed to by *settings* and returns a pointer to that structure.

Refer to the *ctools.h* file for a description of the fields in the *prot_settings* structure.

Refer to the **Overview of Functions** section for detailed information on communication protocols.

See Also

set_protocol

Example

This program displays the protocol configuration for **com1**.

```
#include <ctools.h>

void main(void)
{
	struct prot_settings settings;

	get_protocol(com1, &settings);
	printf("Type:     %d\r\n", settings.type);
	printf("Station:  %d\r\n", settings.station);
	printf("Priority: %d\r\n", settings.priority);
}
```

getProtocolSettings

Get Protocol Extended Addressing Configuration

Syntax

```
#include <ctools.h>
BOOLEAN getProtocolSettings(
FILE * stream,
PROTOCOL_SETTINGS * settings
);
```

Description

The getProtocolSettings function reads the protocol parameters for a serial port. This function supports extended addressing.

The function has two parameters: *stream* is one of com1, com2, com3 or com4; and *settings,* a pointer to a PROTOCOL_SETTINGS structure. Refer to the description of the structure for an explanation of the parameters.

The function returns **TRUE** if the structure was changed. It returns **FALSE** if the stream is not valid.

Notes

Extended addressing is available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

Refer to the ***TeleBUS Protocols User Manual*** section for detailed information on communication protocols.

See Also

setProtocolSettings, get_protocol

Example

```
This program displays the protocol configuration for com1.
#include <ctools.h>

void main(void)
{
	PROTOCOL_SETTINGS settings;

	if (getProtocolSettings(com1, &settings)
	{
		printf("Type: %d\r\n", settings.type);
		printf("Station: %d\r\n", settings.station);
		printf("Address Mode: %d\r\n", settings.mode);
		printf("SF Messaging: %d\r\n", settings.SFMessaging);
		printf("Priority: %d\r\n", settings.priority);
	}
	else
	{
```

```
                printf("Serial port is not valid\r\n");
        }
}
```

getProtocolSettingsEx

Reads extended protocol settings for a serial port.

Syntax

```
#include <ctools.h>
BOOLEAN getProtocolSettingsEx(
       FILE * stream,
       PROTOCOL_SETTINGS_EX * pSettings
       );
```

Description

The setProtocolSettingsEx function sets protocol parameters for a serial port. This function supports extended addressing and Enron Modbus parameters.

The function has two arguments:

- stream specifies the serial port. It is one of com1, com2, com3 or com4.
- pSettings is a pointer to a PROTOCOL_SETTINGS_EX structure. Refer to the description of the structure for an explanation of the parameters.

The function returns TRUE if the settings were retrieved. It returns FALSE if the stream is not valid.

Notes

Extended addressing and the Enron Modbus station are available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

See Also

setProtocolSettingsEx

Example

This program displays the protocol configuration for com1.

```
#include <ctools.h>
void main(void)
{
       PROTOCOL_SETTINGS_EX settings;
       if (getProtocolSettingsEx(com1, &settings)
       {
              printf("Type: %d\r\n", settings.type);
              printf("Station: %d\r\n", settings.station);
              printf("Address Mode: %d\r\n", settings.mode);
              printf("SF: %d\r\n", settings.SFMessaging);
              printf("Priority: %d\r\n", settings.priority);
              printf("Enron: %d\r\n", settings.enronEnabled);
              printf("Enron station: %d\r\n",
                     settings.enronStation);
       }
```

```
		else
		{
			printf("Serial port is not valid\r\n");
		}
	}
```

get_protocol_status

Get Protocol Information

Syntax

```
#include <ctools.h>
struct prot_status get_protocol_status(FILE *stream);
```

Description

The **get_protocol_status** function returns the protocol error and message counters for *stream*. If *stream* does not point to a valid serial port the function has no effect.

Refer to the **Overview of Functions** section for detailed information on communication protocols.

See Also

Error! Reference source not found.

Example

This program displays the checksum error counter for **com2**.

```
#include <ctools.h>

void main(void)
{
	struct prot_status status;

	status = get_protocol_status(com2);
	printf("Checksum: %d\r\n",
		status.checksum_errors);
}
```

getSFMapping

Read Translation Table Mapping Control

Syntax

```
#include <ctools.h>
unsigned getSFMapping(void);
```

Description

The **getSFMapping** and **setSFMapping** functions no longer perform any useful function but are maintained as stubs for backward compatibility. Include the CNFG_StoreAndForward module in the Register Assignment to assign a store and forward table to the I/O database.

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

See Also

addRegAssignment

getSFTranslation

Read Store and Forward Translation

Syntax

```
#include <ctools.h>
struct SFTranslation getSFTranslation(unsigned index);
```

Description

The **getSFTranslation** function returns the entry at *index* in the store and forward address translation table. If *index* is invalid, a disabled table entry is returned.

The function returns a SFTranslation structure. It is described in the **Structures and Types** section.

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

See Also

setSFTranslation, clearSFTranslationTable, checkSFTranslationTable

Example

See the example for the **setSFTranslation** function.

get_status

Get Serial Port Status

Syntax

```
#include <ctools.h>
struct pstatus *get_status(FILE *stream, struct pstatus *status);
```

Description

The **get_status** function returns serial port error counters, I/O lines status and I/O driver buffer information for *stream*. If *stream* does not point to a valid serial port the function has no effect. *status* needs to point to a valid serial port status structure, *pstatus*.

The **get_status** function copies the serial port status into the structure pointed to by *status* and returns a pointer to that structure *settings*.

Refer to the **Overview of Functions** section for detailed information on serial ports.

See Also

clear_errors

Example

This program displays the framing and parity errors for **com1**.

```
#include <ctools.h>

void main(void)
{
	struct pstatus status;

	get_status(com1, &status);
	printf("Framing: %d\r\n", status.framing);
	printf("Parity:  %d\r\n", status.parity);
}
```

getStatusBit

Read Bits in Controller Status Code

Syntax

```
#include <ctools.h>
unsigned getStatusBit(unsigned bitMask);
```

Description

The **getStatusBit** function returns the values of the bits indicated by *bitMask* in the controller status code.

See Also

setStatusBit, setStatus, clearStatusBit

getTaskInfo

Get Information on a Task

Syntax

```
#include <ctools.h>
TASKINFO getTaskInfo(unsigned taskID);
```

Description

The **getTaskInfo** function returns information about the task specified by *taskID*. If *taskID* is 0 the function returns information about the current task.

Notes

If the specified task ID does not identify a valid task, all fields in the return data are set to zero. The calling function should check the taskID field in the TASKINFO structure: if it is zero the remaining information is not valid.

Refer to the **Structures and Types** section for a description of the fields in the TASKINFO structure.

Example

The following program displays information about all valid tasks.

```
#include <string.h>
#include <ctools.h>

void main(void)
{
	struct prot_settings settings;
	TASKINFO taskStatus;
	unsigned task;
	char state[6][20];
	char type[2][20];

	/* Set up state strings */
	strcpy(state[TS_READY], "Ready");
	strcpy(state[TS_EXECUTING], "Executing");
	strcpy(state[TS_WAIT_ENVELOPE], "Waiting for Envelope");
	strcpy(state[TS_WAIT_EVENT], "Waiting for Event");
	strcpy(state[TS_WAIT_MESSAGE], "Waiting for Message");
	strcpy(state[TS_WAIT_RESOURCE], "Waiting for Resource");

	/* Set up type strings */
	strcpy(type[APPLICATION], "Application");
	strcpy(type[SYSTEM], "System");

	/* Disable the protocol on serial port 1 */
	settings.type = NO_PROTOCOL;
	settings.station = 1;
	settings.priority = 3;
	settings.SFMessaging = FALSE;
	request_resource(IO_SYSTEM);
```

```
        set_protocol(com1, &settings);
        release_resource(IO_SYSTEM);

        /* display information about all tasks */
        for (task = 0; task <= RTOS_TASKS; task++)
        {
                taskStatus = getTaskInfo(task);
                if (taskStatus.taskID != 0)
                {
                        /* show information for valid task */
                        fprintf(com1, "\r\n\r\nInformation about task
%d:\r\n", task);
                        fprintf(com1, "    Task ID:  %d\r\n",
taskStatus.taskID);
                        fprintf(com1, "    Priority: %d\r\n",
taskStatus.priority);
                        fprintf(com1, "    Status:   %s\r\n",
state[taskStatus.status]);
                        if (taskStatus.status == TS_WAIT_EVENT)
                        {
                                fprintf(com1, "    Event:    %d\r\n",
taskStatus.requirement);
                        }
                        if (taskStatus.status == TS_WAIT_RESOURCE)
                        {
                                fprintf(com1, "    Resource: %d\r\n",
taskStatus.requirement);
                        }
                        fprintf(com1, "    Error:    %d\r\n",
taskStatus.error);
                        fprintf(com1, "    Type:     %s\r\n",
type[taskStatus.type]);
                }
        }

        while (TRUE)
        {
                /* Allow other tasks to execute */
                release_processor();
        }
}
```

getVersion

Get Firmware Version Information

Syntax

```
#include <ctools.h>
VERSION getVersion(void);
```

Description

The **getVersion** function obtains firmware version information. It returns a VERSION structure. Refer to the **Structures and Types** section for a description of the fields in the VERSION structure.

Notes

The version information can be used to adapt a program to a specific type of controller or version of firmware. For example, a bug work-around could be executed only if older firmware is detected.

Example

This program displays the version information.

```
#include <ctools.h>
void main(void)
{
	struct prot_settings settings;
	VERSION versionInfo;

	/* Disable the protocol on serial port 1 */
	settings.type =        NO_PROTOCOL;
	settings.station =     1;
	settings.priority =    3;
	settings.SFMessaging = FALSE;
	request_resource(IO_SYSTEM);
	set_protocol(com1, &settings);
	release_resource(IO_SYSTEM);

	/* Display the ROM version information */
	versionInfo = getVersion();
	fprintf(com1, "\r\nFirmware Information\r\n");

fprintf(com1, "  Controller type:  %d\r\n", versionInfo.controller
& BASE_TYPE_MASK);
	fprintf(com1, "  Firmware version: %d\r\n",
versionInfo.version);
	fprintf(com1, "  Creation date:    %s\r\n",
versionInfo.date);
	fprintf(com1, "  Copyright:        %s\r\n",
versionInfo.copyright);
}
```

getWakeSource

Gets Conditions for Waking from Sleep Mode

Syntax

```
#include <ctools.h>
unsigned getWakeSource(void);
```

Description

The **getWakeSource** function returns a bit mask of the active wake up sources. Valid wake up sources are listed below.

- WS_REAL_TIME_CLOCK
- WS_INTERRUPT_INPUT
- WS_LED_POWER_SWITCH
- WS_COUNTER_0_OVERFLOW
- WS_COUNTER_1_OVERFLOW

WS_COUNTER_2_OVERFLOW

See Also

setWakeSource, sleep

Example

The following code fragment displays the enabled wake up sources.

```
unsigned enabled;

enabled = getWakeSource();
fputs("Enabled wake up sources:\r\n", com1);
if (enabled & WS_REAL_TIME_CLOCK)
       fputs("  Real Time Clock\r\n", com1);
if (enabled & WS_INTERRUPT_INPUT)
       fputs("  Interrupt Input\r\n", com1);
if (enabled & WS_LED_POWER_SWITCH)
       fputs("  LED Power Switch\r\n", com1);
if (enabled & WS_COUNTER_0_OVERFLOW)
       fputs("  Counter 0 Overflow\r\n", com1);
if (enabled & WS_COUNTER_1_OVERFLOW)
       fputs("  Counter 1 Overflow\r\n", com1);
if (enabled & WS_COUNTER_2_OVERFLOW)
       fputs("  Counter 2 Overflow\r\n", com1);
```

hartIO

Read and Write 5904 HART Interface Module

Syntax

```
#include <ctools.h>
BOOLEAN hartIO(unsigned module);
```

Description

This function reads the specified 5904 interface module. It checks if a response has been received and if a corresponding command has been sent. If so, the response to the command is processed.

This function writes the specified 5904 interface module. It checks if there is a new command to send. If so, this command is written to the 5904 interface.

The function has one parameter: the module number of the 5904 interface (0 to 3).

The function returns TRUE if the 5904 interface responded and FALSE if it did not or if the module number is not valid.

Notes

This function is called automatically if the 5904 module is in the register assignment. Use this function to implement communication with the 5904 if register assignment is not used.

See Also

hartSetConfiguration, hartGetConfiguration, hartCommand

hartIOFromDbase

Read and Write 5904 HART Interface Module with Settings from Database

Syntax

```
#include <ctools.h>
BOOLEAN hartIOFromDbase(unsigned module, unsigned firstRegister);
```

Description

This function reads the specified 5904 interface module. It checks if a response has been received and if a corresponding command has been sent. If so, the response to the command is processed.

This function writes configuration and commands to the specified 5904 interface module. Configuration data is read from the I/O database. It checks if there is a new command to send. If so, this command is written to the 5904 interface.

The function has two parameters: the module number of the 5904 interface (0 to 3); and the address of the first register of a group of four containing the HART interface configuration.

The function returns TRUE if the 5904 interface responded and FALSE if it did not or if the module number is not valid or there is an error in the settings.

See Also

hartIO, hartSetConfiguration

hartCommand

Send Command using HART Interface Module

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand(
       unsigned module,
       HART_DEVICE * const device,
       HART_COMMAND * const command,
       void (* processResponse)( unsigned,
       HART_RESPONSE)
       );
```

Description

This function sends a command to a HART slave device using a HART interface module. This function can be used to implement HART commands not provided by the Network Layer API.

The function has four parameters. The first is the module number of the 5904 interface (0 to 3). The second is the device to which the command is to be sent.

The third parameter is a structure describing the command to send. This contains the command number, and the data field of the HART message. See the HART protocol documentation for your device for details.

The fourth parameter is a pointer to a function that will process the response. This function is called when a response to the command is received by the HART interface. The function is defined as follows:

void *function_name*(HART_RESPONSE response)

The single parameter is a structure containing the response code and the data field from the message.

The function returns TRUE if the 5904 interface responded and FALSE if it did not or if the module number is not valid or there is an error in the command.

Notes

The function returns immediately after the command is sent. The calling program needs to wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

A program needs to initialize the link before executing any other commands.

The function determines if long or short addressing is to be used by the command number. Long addressing is used for all commands except commands 0 and 11.

The functions hartCommand0, hartCommand1, etc. are used to send commands provided by the Network Layer.

See Also

hartStatus, hartSetConfiguration, hartCommand0, hartCommand1

hartCommand0

Read Unique Identifier

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand0(unsigned module, unsigned address,
HART_DEVICE * const device);
```

Description

This function reads the unique identifier of a HART device using command 0 with a short-form address. This is a link initialization function.

The function has three parameters: the module-number of the 5904 module (0 to 3); the short-form address of the HART device (0 to 15); and a pointer to a HART_DEVICE structure. The information read by command 0 is written into the HART_DEVICE structure when the response is received by the 5904 interface.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid, or if the device address is invalid.

Notes

The function returns immediately after the command is sent. The calling program needs to wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

A program needs to initialize the link before executing any other commands.

See Also

hartCommand11, hartStatus, hartSetConfiguration

hartCommand1

Read Primary Variable

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand1(unsigned module, HART_DEVICE * const device,
HART_VARIABLE * primaryVariable);
```

Description

This function reads the primary variable of a HART device using command 1.

The function has three parameters: the module-number of the 5904 module (0 to 3); the device to be read; and a pointer to the primary variable. The variable pointed to by primaryVariable is updated when the response is received by the 5904 interface.

The primaryVariable needs to be a static modular or global variable. A primaryVariable should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure needs to be initialized using hartCommand0 or hartCommand11.

The function returns immediately after the command is sent. The calling program needs to wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

The code field of the HART_VARIABLE structure not changed. Command 1 does not return a variable code.

See Also

hartCommand2, hartStatus, hartSetConfiguration

hartCommand2

Read Primary Variable Current and Percent of Range

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand2(unsigned module, HART_DEVICE * const device,
HART_VARIABLE * pvCurrent, HART_VARIABLE * pvPercent);
```

Description

This function reads the primary variable (PV), as current and percent of range, of a HART device using command 2.

The function has four parameters: the module-number of the 5904 module (0 to 3); the device to be read; a pointer to the PV current variable; and a pointer to the PV percent variable. The pvCurrent and pvPercent variables are updated when the response is received by the 5904 interface.

The pvCurrent and pvPercent variables need to be static modular or global variables. A pvCurrent and pvPercent variable should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure needs to be initialized using hartCommand0 or hartCommand11.

The function returns immediately after the command is sent. The calling program needs to wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

The code field of both HART_VARIABLE structures is not changed. The response from the HART device to command 2 does not include variable codes.

The units field of the pvCurrent variable is set to 39 (units = mA). The units field of the pvPercent variable is set to 57 (units = percent). The response from the HART device to command 2 does not include units.

See Also

hartCommand1, hartStatus, hartSetConfiguration

hartCommand3

Read Primary Variable Current and Dynamic Variables

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand3(unsigned module, HART_DEVICE * const device,
HART_VARIABLE * variables);
```

Description

This function reads dynamic variables and primary variable current from a HART device using command 3.

The function has three parameters: the module number of the 5904 module (0 to 3); the device to be read; and a pointer to an array of five HART_VARIABLE structures.

The variables array needs to be static modular or global variables. An array of variables should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The variables array is updated when the response is received by the 5904 interface as follows.

Variable	Contains
variables[0]	primary variable current
variables[1]	primary variable
variables[2]	secondary variable
variables[3]	tertiary variable
variables[4]	fourth variable

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure needs to be initialized using hartCommand0 or hartCommand11.

The function returns immediately after the command is sent. The calling program needs to wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

Not all devices return primary, secondary, tertiary and fourth variables. If the device does not support a variable, zero is written into the value and units code for that variable.

The code field of both HART_VARIABLE structures is not changed. The response from the HART device to command 3 does not include variable codes.

The units field of variable[0] is set to 39 (units = mA). The response from the HART device to command 3 does not include units.

See Also

hartCommand33**, hartStatus, hartSetConfiguration**

hartCommand11

Read Unique Identifier Associated with Tag

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand11(unsigned module, char * deviceTag,
HART_DEVICE * device);
```

Description

This function reads the unique identifier of a HART device using command 11. This is a link initialization function.

The function has three parameters: the module number of the 5904 module (0 to 3); a pointer to a null terminated string containing the tag of the HART device; and a pointer to a HART_DEVICE structure. The information read by command 11 is written into the HART_DEVICE structure when the response is received by the 5904 interface.

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The function returns immediately after the command is sent. The calling program needs to wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

A program needs to initialize the link before executing any other commands.

See Also

hartCommand0, hartStatus, hartSetConfiguration

hartCommand33

Read Transmitter Variables

Syntax

```
#include <ctools.h>
BOOLEAN hartCommand33(unsigned module, HART_DEVICE * const device,
unsigned variableCode[4], HART_VARIABLE * variables);
```

Description

This function reads selected variables from a HART device using command 33.

The function has four parameters: the module number of the 5904 module (0 to 3); the device to be read; an array of codes; and a pointer to an array of four HART_VARIABLE structures.

The variables array needs to be static modular or global variables. An array of variables should be declared for each HART I/O module in use. A local variable or dynamically allocated variable may not be used because a late command response received after the variable is freed will write data over the freed variable space.

The variableCode array specifies which variables are to be read from the transmitter. Consult the documentation for the transmitter for valid values.

The variables array is updated when the response is received by the 5904 interface as follows.

Variable	Contains
variables[0]	transmitter variable, code and units specified by variableCode[0]
variables[1]	transmitter variable, code and units specified by variableCode[1]
variables[2]	transmitter variable, code and units specified by variableCode[2]
variables[3]	transmitter variable, code and units specified by variableCode[3]

The function returns TRUE if the command was sent. The function returns FALSE if the module number is invalid.

Notes

The HART_DEVICE structure needs to be initialized using hartCommand0 or hartCommand11.

The pointer variables needs to point to an array with at least four elements.

The function returns immediately after the command is sent. The calling program needs to wait for the response to be received. Use the hartStatus command to read the status of the command.

The number of attempts and the number of preambles sent are set with the hartSetConfiguration command.

The function always requests four variables and expects four variables in the response.

See Also

hartCommand3, hartStatus, hartSetConfiguration

hartStatus

Return Status of Last HART Command Sent

Syntax

```
#include <ctools.h>
BOOLEAN hartStatus(unsigned module, HART_RESULT * status, unsigned
* code);
```

Description

This function returns the status of the last HART command sent by a 5904 module (0 to 3). Use this function to determine if a response has been received to a command sent.

The function has three parameters: the module number of the 5904 module; a pointer to the status variable; and a pointer to the additional status code variable. The status and code variables are updated with the following information.

Result	Status	code
HART interface module is not communicating	HR_NoModuleResponse	not used
Command ready to be sent	HR_CommandPending	not used
Command sent to device	HR_CommandSent	current attempt number
Response received	HR_Response	response code from HART device (see Notes)
No valid response received after all attempts made	HR_NoResponse	0=no response from HART device. Other = error response code from HART device (see Notes)
HART interface module is not ready to transmit	HR_WaitTransmit	not used

The function returns TRUE if the status was read. The function returns FALSE if the module number is invalid.

Notes

The response code from the HART device contains communication error and status information. The information varies by device, but there are some common values.

- If bit 7 of the high byte is set, the high byte contains a communication error summary. This field is bit-mapped. The table shows the meaning of each bit as defined by the HART protocol specifications. Consult the documentation for the HART device for more information.

Bit	Description
6	vertical parity error
5	overrun error
4	framing error
3	longitudinal parity error
2	reserved – always 0
1	buffer overflow
0	Undefined

- If bit 7 of the high byte is cleared, the high byte contains a command response summary. The table shows common values. Other values may be defined for specific commands. Consult the documentation for the HART device.

Code	Description
32	Busy – the device is performing a function that cannot be interrupted by this command
64	Command not Implemented – the command is not defined for this device.

- The low byte contains the field device status. This field is bit-mapped. The table shows the meaning of each bit as defined by the HART protocol specifications. Consult the documentation for the HART device for more information.

Bit	Description
7	field device malfunction
6	configuration changed
5	cold start
4	more status available (use command 48 to read)
3	primary variable analog output fixed
2	primary variable analog output saturated
1	non-primary variable out of limits
0	primary variable out of limits

See Also

hartSetConfiguration

hartGetConfiguration

Read HART Module Settings

Syntax

```
#include <ctools.h>
BOOLEAN hartGetConfiguration(unsigned module, HART_SETTINGS *
settings);
```

Description

This function returns the configuration settings of a 5904 module.

The function has two parameters: the module number of the 5904 module (0 to 3); and a pointer to the settings structure.

The function returns TRUE if the settings were read. The function returns FALSE if the module number is invalid.

See Also

hartSetConfiguration

hartSetConfiguration

Write HART Module Settings

Syntax

```
#include <ctools.h>
BOOLEAN hartSetConfiguration(unsigned module, HART_SETTINGS
settings);
```

Description

This function writes configuration settings to a 5904 module.

The function has two parameters: the module number of the 5904 module (0 to 3); and a settings structure.

The function returns TRUE if the settings were written. The function returns FALSE if the module number or the settings are invalid.

Notes

The configuration settings are stored in the EEPROM_RUN section of the EEPROM. The user-defined settings are used when the controller is reset in the RUN mode. Default settings are used when the controller is reset in the SERVICE or COLD BOOT modes.

If a **CNFG 5904 HART Interface** module is in the register assignment, forced registers from it take precedence over the settings supplied here.

See Also

hartGetConfiguration

hartPackString

Convert String to HART Packed String

Syntax

```
#include <ctools.h>
void hartPackString(char * pPackedString, const char * pString,
unsigned sizePackedString);
```

Description

This function stores an ASCII string into a HART packed ASCII string.

The function has three parameters: a pointer to a packed array; a pointer to an unpacked array; and the size of the packed array. The packed array needs to be a multiple of three in size. The unpacked array needs to be a multiple of four in size. It should be padded with spaces at the end if the string is not long enough.

The function has no return value.

See Also

hartUnpackString

hartUnpackString

Convert HART Packed String to String

Syntax

```
#include <ctools.h>
void hartUnpackString(char * pString, const char * pPackedString,
unsigned sizePackedString);
```

Description

This function unpacks a HART packed ASCII string into a normal ASCII string.

The function has three parameters: a pointer to an unpacked array; a pointer to a packed array; and the size of the packed array. The packed array needs to be a multiple of three in size. The unpacked array needs to be a multiple of four in size.

The function has no return value.

See Also

hartPackString

install_handler

Install Serial Port Handler

Syntax

```
#include <ctools.h>
void install_handler(FILE *stream, void *function(unsigned,
unsigned));
```

Description

The **install_handler** function installs a serial port character handler function. The serial port driver calls this function each time it receives a character. If *stream* does not point to a valid serial port the function has no effect.

function specifies the handler function, which takes two arguments. The first argument is the received character. The second argument is an error flag. A non-zero value indicates an error. If *function* is **NULL**, the default handler for the port is installed. The default handler does nothing.

Notes

The **install_handler** function can be used to write custom communication protocols.

The handler is called at the completion of the receiver interrupt handler. RTOS calls (see functions listed in the section *Real Time Operating System Functions* at the start of this chapter) may not be made within the interrupt handler, with one exception. The **interrupt_signal_event** RTOS call can be used to signal events.

To optimize performance, minimize the length of messages on com3 and com4. Examples of recommended uses for com3 and com4 are for local operator display terminals, and for programming and diagnostics using the Telepace program.

Example

```
#include <ctools.h>

#define CHAR_RECEIVED 11


/* --------------------------------------------
   signal

   This routine signals an event when a character
   is received on com1. If there is an error, the
   character is ignored.
   -------------------------------------------- */

void signal(unsigned character, unsigned error)
{
       if (error == 0)
              interrupt_signal_event( CHAR_RECEIVED );
```

```
        character;
}

/* --------------------------------------------
   main

   This program displays all characters received
   on com1 using an installed handler to signal
   the reception of a character.
   -------------------------------------------- */

void main(void)
{
        struct prot_settings protocolSettings;
        int character;

        /* Disable protocol */
        get_protocol(com1, &protocolSettings);
        protocolSettings.type = NO_PROTOCOL;
        request_resource(IO_SYSTEM);
        set_protocol(com1, &protocolSettings);
        release_resource(IO_SYSTEM);

        /* Enable character handler */
        install_handler(com1, signal);

        /* Print each character as it is recevied */
        while (TRUE)
        {
                wait_event(CHAR_RECEIVED);
                character = fgetc(com1);
                fputs("character: ", com1);
                fputc(character, com1);
                fputs("\r\n", com1);
        }
}
```

installClockHandler

Install Handler for Real Time Clock

Syntax

```
#include <ctools.h>
void installClockHandler(void (*function)(void));
```

Description

The **installClockHandler** function installs a real time clock alarm handler function. The real time clock alarm function calls this function each time a real time clock alarm occurs.

function specifies the handler function. If *function* is **NULL**, the handler is disabled.

Notes

RTOS calls (see functions listed in the section *Real Time Operating System Functions* at the start of this chapter) may not be made within the interrupt handler, with one exception. The **interrupt_signal_event** RTOS call can be used to signal events.

See Also

setClockAlarm

Example

```
/* --------------------------------------------
   This program demonstrates how to call a
   function at a specific time of day.
   -------------------------------------------- */

#include <ctools.h>

#define       ALARM_EVENT         20

/* --------------------------------------------
       This function signals an event when the alarm
       occurs.
-------------------------------------------- */
void alarmHandler(void)
{
       interrupt_signal_event( ALARM_EVENT );
}

/* --------------------------------------------
   This task processes alarms signaled by the
       clock handler
-------------------------------------------- */
void processAlarms(void)
{
```

```
        while(TRUE)
        {
                wait_event(ALARM_EVENT);

                /* Reset the alarm for the next day */
                request_resource(IO_SYSTEM);
                resetClockAlarm();
                release_resource(IO_SYSTEM);

                fprintf(com1, "It's quitting time!\r\n");
        }
}

void main(void)
{
        struct prot_settings settings;
        ALARM_SETTING alarm;

        /* Disable the protocol on serial port 1 */
        settings.type = NO_PROTOCOL;
        settings.station = 1;
        settings.priority = 3;
        settings.SFMessaging = FALSE;
        request_resource(IO_SYSTEM);
        set_protocol(com1, &settings);
        release_resource(IO_SYSTEM);

        /* Install clock handler function */
        installClockHandler(alarmHandler);

        /* Create task for processing alarm events */
        create_task(processAlarms, 3, APPLICATION, 4);

        /* Set real time clock alarm */
        alarm.type   = AT_ABSOLUTE;
        alarm.hour   = 16;
        alarm.minute = 0;
        alarm.second = 0;

        request_resource(IO_SYSTEM);
        setClockAlarm(alarm);
        release_resource(IO_SYSTEM);

        while(TRUE)
        {
                /* body of main task loop */

                /* other processing code */

                /* Allow other tasks to execute */
                release_processor();
        }
}
```

installExitHandler

Install Handler Called when Task Ends

Syntax

```
#include <ctools.h>
unsigned installExitHandler(unsigned taskID, void (*function)(void));
```

Description

The **installExitHandler** function defines a function that is called when the task, specified by *taskID*, is ended. *function* specifies the handler function. If *function* is **NULL**, the handler is disabled.

Notes

The exit handler function will be called when:

- the task is ended by the end_task function
- the end_application function is executed and the function is an APPLICATION type function
- the program is stopped from the Telepace program and the task is an APPLICATION type function
- the C program is erased by the Telepace program.

The exit handler function is not called if power to the controller is removed. In this case execution stops when power is removed. The application program starts from the beginning when power is reapplied.

RTOS functions cannot be called from the exit handler.

Example

See the example for **startTimedEvent**.

installModbusHandler

Install User Defined Modbus Handler

Syntax

```
#include <ctools.h>
void installModbusHandler(
unsigned (* handler)(unsigned char *, unsigned,
                     unsigned char *, unsigned *)
       );
```

Description

The installModbusHandler function allows user-defined extensions to standard Modbus protocol. This function specifies a function to be called when a Modbus message is received for the station, but is not understood by the standard Modbus protocol. The installed handler function is called only if the message is addressed to the station, and the message checksum is correct.

The function has one parameter: a pointer to a function to handle the messages. See the section **Handler Function** for a description of the function and it's parameters. If the pointer is NULL, no function is called for non-standard messages.

The function has no return value.

Notes

This function is used to create a user-defined extension to the standard Modbus protocol.

Call this function with the NULL pointer to disable processing of non-standard Modbus messages. This needs to be done when the application program is ended with an exit handler. Use the installExitHandler function to install the exit handler.

If the Modbus handler is not disabled within an exit handler, it will remain installed and continue to operate until the controller power is cycled. Changing the protocol type or *Erasing the C Program* from Telepace Initialize dialog will not remove the Modbus handler. If the handler is located in a RAM-based application and left enabled while a different C application is downloaded, the original handler will be corrupted.

See Also

installExitHandler, Handler Function

Handler Function

User Specified Handler Function

The handler function is a user-specified function that handles processing of Modbus messages not recognized by the protocol. The function can have any name; *handler* is used in the description below.

Syntax

```
#include <ctools.h>
unsigned handler(
       unsigned char * message,
       unsigned messageLength,
       unsigned char * response,
       unsigned * responseLength
       );
```

Description

This function *handler* is a user-defined handler for processing Modbus messages. The function is called for each Modbus message with a function code that is not recognized by the standard Modbus protocol.

The *handler* function should process the message string and create a response string. IF the message is not understood, one of the error codes should be returned.

The function has four parameters.

- The *message* parameter is a pointer to the first character of the received message. The first character of the message is the function code. The format of the data after the function code is defined by the function code.
- The *messageLength* parameter is the number of characters in the message.
- The *response* parameter is a pointer to the first character of a buffer to hold the response. The function should write the response into this buffer. The buffer is 253 characters long. The first character of the buffer is the function code of the message. The format of the data after the function code is defined by the function code.
- The *responseLength* parameter is a pointer to the length of the response. The function should set the length of the response using this pointer. The length is the number of characters placed into the response buffer.

The function must return one of four values. The first causes a normal response to be sent. The others cause an exception response to be sent.

- NORMAL indicates the response and responseLength have been set to valid values. The Modbus protocol will add the station address and checksum to this string and transmit the reply to the master station.
- ILLEGAL_FUNCTION indicates the function code in the message was not understood. The *handler* function must return this value for all function codes

it does not process. The Modbus protocol will return an Illegal Function exception response.

- ILLEGAL_DATA_ADDRESS indicates the function code in the message was understood, but that the command referenced an address that is not valid. The Modbus protocol will return an Illegal Data Address exception response.
- ILLEGAL_DATA_VALUE indicates the function code in the message was understood, but that the command included data that is not valid. The Modbus protocol will return an Illegal Data Address exception response.

Function Codes Used

The following function codes are currently used by the TeleBUS Modbus-compatible protocol. All other function codes are available for use. For maximum compatibility with other Modbus and Modbus-compatible devices it is recommended that codes in the user-defined function code range be used first.

Code	Type	Description
1	Modbus standard	Read coil registers from I/O database
2	Modbus standard	Read status registers from I/O database
3	Modbus standard	Read holding registers from I/O database
4	Modbus standard	Read input registers from I/O database
5	Modbus standard	Write a single coil register
6	Modbus standard	Write a single holding register
7	Modbus standard	Read exception status
15	Modbus standard	Write multiple coil registers
16	Modbus standard	Write multiple holding registers
17	Modbus standard	Report slave identification string
65	TeleBUS extension	Used by Telepace
66	TeleBUS extension	Used by Telepace
67	TeleBUS extension	Used by Telepace
68	TeleBUS extension	Used by Telepace
69	TeleBUS extension	Used by Telepace
70	TeleBUS extension	Used by Telepace

Notes

One *handler* function is used for all serial ports. Only one port will be active at any time. Therefore, the function does not have to be re-entrant.

The *handler* function is called from the Modbus protocol task. This task may pre-empt the execution of another task. If there are shared resources, the *handler* function needs to request and release the appropriate resources for proper operation.

The station address is not included in the message or response string. It will be added to the response string before sending the reply.

The checksum is not included in the message or the response string. It will be added to the response string before sending the reply.

The maximum size of the response string is 253 bytes. If a longer response length is returned, the Modbus protocol will report an ILLEGAL_DATA_VALUE exception. The response will not be returned.

See Also

installModbusHandler

Example

```
/* ------------------------------------------------
   handler.c

   This is a sample program for the InstallModbusHandler function.
This sample  program uses function code 71 to demonstrate a
simple method for using the installModbusHandler function.
   When the handler is installed Modbus ASCII  messages using
function code 71 that are  received on com2 of the controller will
   be processed as shown in the program text.

   To turn on digital output 00001:
   From a terminal send the ASCII command     :014701B7
   Where;
       01 is the station address
       47 is the function code in hex
       01 is the command for the function code
       B7 is the message checksum

   To turn off digital output 00001:
   From a terminal send the ASCII command     :014700B8
   Where;
       01 is the station address
       47 is the function code in hex
       00 is the command for the function code
       B8 is the message checksum
   --------------------------------------------- */
#include <ctools.h>

static unsigned myModbusHandler(
       unsigned char * message,
       unsigned     messageLength,
       unsigned char * response,
       unsigned * responseLength
       )
{
       unsigned char * pMessage;
       unsigned char * pResponse;

       pMessage = message;

       if (*pMessage == 71)
```

```
        {
                /* Action for command data */
                pMessage++;

             if (*pMessage == 0)
                {
                        request_resource(IO_SYSTEM);
                        setdbase(MODBUS, 1, 0);
                        release_resource(IO_SYSTEM);

                        pResponse = response;

                        *pResponse    = 71;
                        pResponse++;
                        *pResponse    = 'O';
                        pResponse++;
                        *pResponse    = 'F';
                        pResponse++;
                        *pResponse    = 'F';
                        pResponse++;

                        *responseLength = 4;

                        return NORMAL;
                }

                if (*pMessage == 1)
                {
                        request_resource(IO_SYSTEM);
                        setdbase(MODBUS, 1, 1);
                        release_resource(IO_SYSTEM);


                        pResponse = response;
                        *pResponse    = 71;
                        pResponse++;
                        *pResponse    = 'O';
                        pResponse++;
                        *pResponse    = 'N';
                        pResponse++;
                        *responseLength = 3;

                        return NORMAL;
                }

        }
}

static void shutdown(void)
{
        installModbusHandler(NULL);
}

/* -----------------------------------------------
   main
```

```
   This routine is the modbus slave application.
   Serial port com2 is configured for Modbus ASCII   protocol.
   Register Assignment is configured.
   The modbus handler is installed.
   The exit handler is installed.
   -------------------------------------------- */
void main(void)
{
       TASKINFO taskStatus;

       struct pconfig portSettings;
       struct prot_settings protSettings;

       portSettings.baud          = BAUD9600;
       portSettings.duplex        = FULL;
       portSettings.parity        = NONE;
       portSettings.data_bits     = DATA7;
       portSettings.stop_bits     = STOP1;
       portSettings.flow_rx       = DISABLE;
       portSettings.flow_tx       = DISABLE;
       portSettings.type          = RS232;
       portSettings.timeout       = 600;
       set_port(com2, &portSettings);

       get_protocol(com2, &protSettings);
       protSettings.station     = 1;
       protSettings.type           = MODBUS_ASCII;
       set_protocol(com2, &protSettings);

       /* Configure Register Assignment */
       clearRegAssignment();
       addRegAssignment(DIN_generic8, 0, 10017, 0, 0, 0);
       addRegAssignment(SCADAPack_lowerIO,0, 1, 10001, 30001, 0);
       addRegAssignment(DIAG_protocolStatus,1,31000, 0, 0, 0);

       /* Install Modbus Handler */
       request_resource(IO_SYSTEM);
       installModbusHandler(myModbusHandler);
       release_resource(IO_SYSTEM);

    /* Install Exit Handler */
       taskStatus = getTaskInfo(0);
       installExitHandler(taskStatus.taskID, shutdown);

       while(TRUE)
       {
              release_processor();
       }
}
```

installRTCHandler

Install User Defined Real-Time-Clock Handler

Syntax

```
#include <ctools.h>
void installRTCHandler(
void (* rtchandler)(TIME *now,
                    TIME *new)
       );
```

Description

The installRTCHandler function allows an application program to override Modbus protocol and DNP protocol commands to set the real time clock. This function specifies a function to be called when a Modbus or DNP message is received for the station. The installed handler function is called only if the message is for setting the real time clock.

The function has one parameter: a pointer to a function to handle the messages. See the section **RTCHandler Function** for a full description of the function and its parameters. If the pointer is NULL, no function is called for set the real time clock commands, and the default method is used set the real time clock.

The function has no return value.

Notes

Call this function with the NULL pointer to disable processing of *Set Real Time Clock* messages. This needs to be done when the application program is ended with an exit handler. Use the installExitHandler function to install the exit handler.

If the RTC handler is not disabled within an exit handler, it will remain installed and continue to operate until the controller power is cycled. Changing the protocol type or *Erasing the C Program* from the Telepace Initialize dialog will not remove the handler. If the handler is located in a RAM-based application and left enabled while a different C application is downloaded, the original handler will be corrupted

See Also

RTCHandler Function, installExitHandler

RTCHandler Function

User Specified Real Time Clock Handler Function

The handler function is a user-specified function that handles processing of Modbus messages or DNP messages for setting the real time clock. The function can have any name; *rtchandler* is used in the description below.

Syntax

```
#include <ctools.h>
void rtchandler(
       TIME *now,
       TIME *new
       );
```

Description

This function *rtchandler* is a user-defined handler for processing Modbus messages or DNP messages. The function is called only for messages that set the real time clock.

The *rtchandler* function should set the real time clock to the requested time. If there is a delay before this can be done, the time when the message was received is provided so that a correction to the requested time can be made.

The function has two parameters.

- The *now* parameter is a pointer to the structure containing the time when the message was received.
- The *new* parameter is a pointer to the structure containing the requested time.

The function does not return a value.

Notes

The IO_SYSTEM resource has already been requested before calling this function. If this function calls other functions that require the IO_SYSTEM resource (e.g. setclock), there is no need to request or release the resource.

See Also

installRTCHandler

interruptCounter

Read Interrupt Input Counter

Syntax

```
#include <ctools.h>
unsigned long interruptCounter(unsigned clear);
```

Description

The interruptCounter routine reads the interrupt input as a counter. If *clear* is TRUE the counter is cleared after reading; otherwise if it is FALSE the counter continues to accumulate.

Notes

The interrupt input is located on the 5203 or 5204 controller board. Refer to the ***System Hardware Manual*** for more information on the hardware.

The counter increments on the rising edge of the input signal.

The maximum input frequency that can be counted by the interrupt input is 200 Hz.

See Also

interruptInput, readCounter

interruptInput

Read State of Interrupt Digital Input

Syntax

```
#include <ctools.h>
unsigned interruptInput(void);
```

Description

The **interruptInput** function reads the status of the interrupt input point on the controller. It returns **TRUE** if the input is energized and **FALSE** if it is not.

Notes

The interrupt input can be used as wake up source for the controller or as an additional a digital input. Refer to the ***System Hardware Manual*** for wiring details.

interrupt_signal_event

Signal Event in Interrupt Handler

Syntax

```
#include <ctools.h>
void interrupt_signal_event(unsigned event_number);
```

Description

The **interrupt_signal_event** function is used in an interrupt handler to signal events. The function signals that the *event_number* event has occurred.

If there are tasks waiting for the event, the highest priority task is made ready to execute. Otherwise the event flag is incremented. Up to 255 occurrences of an event will be recorded. The current task is blocked of there is a higher priority task waiting for the event.

Notes

Refer to the **Real Time Operating System** section for more information on events.

This function is only to be used within an interrupt handler.

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in ctools.h. are not valid events for use in an application program.

See Also

signal_event, startTimedEvent, installClockHandler

interval

Set Timer Tick Interval

Syntax

```
#include <ctools.h>
void interval(unsigned timer, unsigned value);
```

Description

The **interval** function sets the tick interval for *timer* to *value*. Tick intervals are measured in multiples of 0.1 second.

If the timer number is invalid, the task's error code is set to **TIMER_BADTIMER**.

Notes

The default timer tick interval is 1/10 second.

See Also

settimer, read_timer_info, check_error

Example

```
Set timer 5 to count 12 seconds using 1.0 s ticks.
interval(5, 10);           /* 1.0 s ticks */
settimer(5, 12);           /* time = 12 seconds */

Set timer 5 to count 12 seconds using 0.1 s ticks.
interval(5, 1);            /* 0.1 s ticks */
settimer(5, 120);          /* time = 12 seconds */
```

ioBusReadByte

Read One Byte from I^2C Slave Device

Syntax

```
#include <ctools.h>
unsigned char ioBusReadByte(void);
```

Description

The **ioBusReadByte** function returns one byte read from an I^2C slave device. The byte is acknowledged by the master receiver. This function can be used multiple times in sequence to read data from a slave device. The last byte read from the slave must be read with the **ioBusReadLastByte** function.

If only one byte is to be read from a device, the **ioBusReadLastByte** function needs to be used instead of this function.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStart, ioBusStop, ioBusReadLastByte, ioBusReadMessage, ioBusSelectForRead ioBusSelectForWrite, ioBusWriteByte, ioBusWriteMessage

Example

```
#include <ctools.h>

void main(void)
{
        unsigned char data[3];
        unsigned char ioBusAddress = 114;

        request_resource(IO_SYSTEM);

        ioBusStart();
        if (ioBusSelectForRead(ioBusAddress))
        {
                data[0] = ioBusReadByte();
                data[1] = ioBusReadByte();
                /* reading the last byte terminates read */
                data[2] = ioBusReadLastByte();
        }
        ioBusStop();

        release_resource(IO_SYSTEM);
}
```

ioBusReadLastByte

Read Last Byte from I^{2}C Slave Device

Syntax

```
#include <ctools.h>
unsigned char ioBusReadLastByte(void);
```

Description

The **ioBusReadLastByte** function returns one byte read from an I^{2}C slave device and terminates reading from the slave. The byte is not acknowledged by the master receiver. This signals to the slave device that the read is complete. This function needs to be used once at the end of a read.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStart, ioBusStop, ioBusReadByte, ioBusReadMessage, ioBusSelectForRead ioBusSelectForWrite, ioBusWriteByte, ioBusWriteMessage

Example

See example for **ioBusReadByte**.

ioBusReadMessage

Read Message from I^2C Slave Device

Syntax

```
#include <ctools.h>
READSTATUS ioBusReadMessage(unsigned address, unsigned
numberBytes, unsigned char *message);
```

Description

The **ioBusReadMessage** function reads a specified number of bytes from an I^2C slave device.

The function issues a START condition, selects the device for reading, reads the specified number of bytes, and issues a STOP condition. It detects if the device cannot be selected and, if so, aborts the read.

The function has three parameters: the *address* of the device; the number of bytes to read, *numberBytes*; and a pointer to a buffer, *message*, capable of holding the data read.

The function returns the status of the read:

Value	Description
RS_success	read was successful
RS_selectFailed	slave device could not be selected

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusWriteMessage, ioBusStart, ioBusStop, ioBusReadByte ioBusReadLastByte, ioBusSelectForRead ioBusSelectForWrite, ioBusWriteByte, ioBusWriteMessage

Example

```
#include <ctools.h>
void main(void)
{
	unsigned char	message[10];
	unsigned char	ioBusAddress = 114;
	READSTATUS	status;
	request_resource(IO_SYSTEM);

	/* Read a 10 byte message from I2C device */
	status = ioBusReadMessage(ioBusAddress, 10,
		message);
	release_resource(IO_SYSTEM);
```

```
        if (status != RS_success)
        {
            fprintf(com1, "I/O error = %d\n\r", status);
        }
}
```

ioBusSelectForRead

Select I^2C Slave Device for Reading

Syntax

```
#include <ctools.h>
unsigned ioBusSelectForRead(unsigned char address);
```

Description

The **ioBusSelectForRead** function selects an I^2C slave device for reading. It writes the slave device address with the read/write bit set to the read state. The function handles the formatting of the address byte.

The function has one parameter, the *address* of the device. It returns TRUE if the write succeeded, that is the byte was acknowledged by the slave. It returns FALSE if the byte was not acknowledged by the slave.

Notes

This function can only be used immediately after a START condition, e.g. **ioBusStart**.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStart, ioBusStop, ioBusReadByte, ioBusReadLastByte, ioBusReadMessage, ioBusSelectForWrite, ioBusWriteByte, ioBusWriteMessage

Example

See example for **ioBusReadByte**.

ioBusSelectForWrite

Select I^2C Slave Device for Writing

Syntax

```
#include <ctools.h>
unsigned ioBusSelectForWrite(unsigned char address);
```

Description

The **ioBusSelectForWrite** function selects an I^2C slave device for writing. It writes the slave device address with the read/write bit set to the write state. The function handles the formatting of the address byte.

The function has one parameter, the *address* of the device. It returns TRUE if the write succeeded, that is the byte was acknowledged by the slave. It returns FALSE if the byte was not acknowledged by the slave.

Notes

This function can only be used immediately after a START condition, e.g. **ioBusStart**.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStart, ioBusStop, ioBusReadByte, ioBusReadLastByte, ioBusReadMessage, ioBusSelectForRead, ioBusWriteByte, ioBusWriteMessage

Example

See example for **ioBusWriteByte**.

ioBusStart

Issue an I²C Bus START Condition

Syntax

```
#include <ctools.h>
void ioBusStart(void);
```

Description

The **ioBusStart** function issues an I^2C bus START condition.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStop, ioBusReadByte, ioBusReadLastByte, ioBusReadMessage, ioBusSelectForRead ioBusSelectForWrite, ioBusWriteByte, ioBusWriteMessage

Example

See example for **ioBusReadByte**.

ioBusStop

Issue an I^2C Bus STOP Condition

Syntax

```
#include <ctools.h>
void ioBusStop(void);
```

Description

The **ioBusStop** function issues an I^2C bus STOP condition.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStart, ioBusReadByte, ioBusReadLastByte, ioBusReadMessage, ioBusSelectForRead ioBusSelectForWrite, ioBusWriteByte, ioBusWriteMessage

Example

See example for **ioBusReadByte**.

ioBusWriteByte

Write One Byte to I^2C Slave Device

Syntax

```
#include <ctools.h>
unsigned ioBusWriteByte(unsigned char byte);
```

Description

The **ioBusWriteByte** function writes one byte to an I^2C slave device and returns the acknowledge signal from the slave. It returns TRUE if the write succeeded, that is the byte was acknowledged by the slave. It returns FALSE if the byte was not acknowledged by the slave.

This function can be used multiple times in sequence to write data to a device.

Notes

ioBusWriteByte can be used to write the address selection byte at the start of an I^2C message; however, the **ioBusSelectForRead** and **ioBusSelectForWrite** functions provide a more convenient interface for doing this.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStart, ioBusStop, ioBusReadByte, ioBusReadLastByte, ioBusReadMessage, ioBusSelectForRead ioBusSelectForWrite, ioBusWriteMessage

Example

```
#include <ctools.h>

void main(void)
{
	unsigned char data[2];
	unsigned char ioBusAddress = 114;

	request_resource(IO_SYSTEM);

	ioBusStart();
	if (ioBusSelectForWrite(ioBusAddress))
	{
		ioBusWriteByte(data[0]);
		ioBusWriteByte(data[1]);
	}
	ioBusStop();

	release_resource(IO_SYSTEM);
}
```

ioBusWriteMessage

Write Message to I^2C Slave Device

Syntax

```
#include <ctools.h>
WRITESTATUS ioBusWriteMessage(unsigned address, unsigned
numberBytes, unsigned char *message);
```

Description

The **ioBusWriteMessage** function writes a specified number of bytes to an I^2C slave device.

The function issues the START condition, selects the device for writing, writes the specified number of bytes, and issues a STOP condition. If the slave does not acknowledge the selection or any data written to it, the write is aborted immediately.

The function has three parameters: the *address* of the device; the number of bytes to write, *numberBytes*; and a pointer to the buffer, *message*, containing the data.

The function returns the status of the write:

Value	Description
WS_success	write was successful
WS_selectFailed	slave could not be selected
WS_noAcknowledge	slave failed to acknowledge data

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioBusStart, ioBusStop, ioBusReadByte, ioBusReadLastByte, ioBusReadMessage, ioBusSelectForRead ioBusSelectForWrite, ioBusWriteByte

Example

```
#include <ctools.h>

void main(void)
{
      unsigned char       message[10];
      unsigned char       ioBusAddress = 114;
      WRITESTATUS         status;

      request_resource(IO_SYSTEM);
```

```
        /* Write a 10 byte message to I2C device */
        status = ioBusWriteMessage(ioBusAddress, 10,
               message);

        release_resource(IO_SYSTEM);

        if (status != WS_success)
        {
               fprintf(com1, "I/O error = %d\n\r", status);
        }
}
```

ioClear

Turn Off all Outputs

Syntax

```
#include <ctools.h>
void ioClear(void);
```

Description

The **ioClear** function turns off all outputs in the current Register Assignment as follows.

- analog outputs are set to 0;
- digital outputs are turned set to 0 (turned off).

If the Register Assignment is empty, all outputs are turned off for all possible I/O modules that exist under the fixed I/O hardware mapping of firmware versions 1.22 or older.

Also, delayed digital I/O actions started by the **pulse**, **pulse_train** and **timeout** functions are canceled.

Notes

Timers referenced by the **pulse**, **pulse_train** and **timeout** functions are set to 0. All other timers are not affected.

The IO_SYSTEM resource needs to be requested before calling this function.

ioDatabaseReset

Initialize I/O Database with Default Values

Syntax

```
#include <ctools.h>
void ioDatabaseReset(void);
```

Description

The **ioDatabaseReset** function resets all I/O database values to their defaults:

- Configuration parameters are reset to default values. All registers assigned to configuration parameters through the Register Assignment are also reset to default values.
- All other registers are set to zero. I/O hardware assigned to these registers through the Register Assignment are also set to zero.
- All forcing is removed.
- Locked variables are unlocked.
- Set all database locations to zero
- Clear real time clock alarm settings
- Clear serial port event counters
- Clear store and forward configuration
- Enable LED power by default and return to default state after 5 minutes
- Set Outputs on Stop settings to Hold
- Set 5904 HART modem configuration for all modems
- Set Modbus/TCP default configuration
- Write new default data to Flash

Notes

This function can be used to restore the controller to its default state. **ioDatabaseReset** has the same effect as selecting the **Initialize Controller** option from the **Initialize** command in the Telepace program.

Use this function carefully as it erases any data stored in the I/O database.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

```
#include <ctools.h>

void main(void)
{
```

```
        /* Power Up Initialization */
        request_resource(IO_SYSTEM);
        ioDatabaseReset();
        release_resource(IO_SYSTEM);

        /* ... the rest of the program */
}
```

ioRead16Din

Read 16 Digital Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead16Din(unsigned moduleAddress, unsigned
startStatusRegister);
```

Description

The **ioRead16Din** function reads any 16 point Digital Input Module at the specified *moduleAddress*. Data is read from all 16 digital inputs and copied to 16 consecutive status registers beginning at *startStatusRegister*.

The function returns FALSE if the *moduleAddress* or *startStatusRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startStatusRegister* is any valid Modbus status register between 10001 and (10000 + NUMSTATUS - 15).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead8Din

Example

This program displays the values of the 16 digital inputs read from a 16 point Digital Input Module at module address 0.

```
#include <ctools.h>

void main(void)
{
	unsigned reg;

	request_resource(IO_SYSTEM);

	/* Read data from digital input module and 	write it to 
I/O database */
	ioRead16Din(0, 10001);
```

```
        /* Print data from I/O database */
        fprintf(com1, "Register          Value");
        for (reg = 10001; reg <= 10016; reg++)
        {
              fprintf(com1, "\n\r%d      ", reg);
              putchar( dbase(MODBUS, reg) ? '1' :'0');
        }

        release_resource(IO_SYSTEM);
}
```

ioRead32Din

Read 32 Digital Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead32Din(
       UINT16 moduleAddress,
       UINT16 startStatusRegister);
```

Description

The ioRead32Din function reads any 32 point Digital Input Module at the specified moduleAddress. Data is read from all the digital inputs and copied to 32 consecutive status registers beginning at startStatusRegister.

The function returns FALSE if the moduleAddress or startStatusRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for moduleAddress is 0 to 15. startStatusRegister is any valid Modbus status register between 10001 and (10001 + NUMSTATUS - 32).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the ctools.lib library. Load this library in you linker command (.cmd) file as shown in the sample file appram.cmd in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead8Din, **ioRead16Din**

Example

This program displays the values of the 32 digital inputs read from a 32 point Digital Input Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Read data from module and write to I/O database */
       ioRead32Din(0, 10001);
```

```
        /* Print data from I/O database */
        fprintf(com1, "Register          Value");
        for (reg = 10001; reg <= 10032; reg++)
        {
                fprintf(com1, "\n\r%d      ", reg);
                putchar( dbase(MODBUS, reg) ? '1' :'0');
        }

        release_resource(IO_SYSTEM);
}
```

ioRead4Ain

Read 4 Analog Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead4Ain(unsigned moduleAddress, unsigned
startInputRegister);
```

Description

The **ioRead4Ain** function reads any 4 point Analog Input Module at the specified *moduleAddress*. Data is read from the 4 analog inputs and copied to 4 consecutive input registers beginning at *startInputRegister*.

The function returns FALSE if the *moduleAddress* or *startInputRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startInputRegister* is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 3).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead8Ain

Example

This program displays the values of the 4 analog inputs read from a 4 point Analog Input Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Read data from digital input module and     write it to
I/O database */
       ioRead4Ain(0, 30001);
```

```
        /* Print data from I/O database */
        fprintf(com1, "Register          Value\n\r");
        for(reg = 30001; reg <= 30004; reg++)
        {
               fprintf(com1, "%d   %d\n\r", reg,
        dbase(MODBUS, reg));
        }

        release_resource(IO_SYSTEM);
}
```

ioRead4Counter

Read 4 Counter Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead4Counter(unsigned moduleAddress, unsigned
startInputRegister);
```

Description

The **ioRead4Counter** function reads any 4 point Counter Input Module at the specified *moduleAddress*. Data is read from the 4 counter inputs and copied to 8 consecutive input registers beginning at *startInputRegister*.

Each counter is a 32 bit number, stored in two input registers. The first register holds the least significant 16 bits of the counter. The second register holds the most significant 16 bits of the counter.

The maximum count is 4,294,967,295. Counters roll back to 0 when the maximum count is exceeded.

The function returns FALSE if the *moduleAddress* or *startInputRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startInputRegister* is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 7).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section **Overview of Functions** for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

This program displays the values of the 4 counter inputs read from a 4 point Counter Input Module at module address 0.

```
#include <ctools.h>

void main(void)
{
        unsigned counter, reg;
        unsigned long       value;

        request_resource(IO_SYSTEM);
```

```
        /* Read data from counter input module and
        write it to I/O database */
        ioRead4Counter(0, 30001);

        /* Print data from I/O database */
        fprintf(com1, "Counter     Value\n\r");
        counter = 0;
        for(reg = 30001; reg <= 30008; reg+=2)
        {
               value = dbase(MODBUS, reg) +
                      ((long) dbase(MODBUS, reg+1)<<16);
               fprintf(com1, "%d   %ld\n\r", counter++,
               value);
        }

        release_resource(IO_SYSTEM);
}
```

ioRead4202Inputs

Read SCADAPack 4202 DR Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead4202Inputs(
       unsigned startStatusRegister,
       unsigned startInputRegister
       );
```

Description

The ioRead4202Inputs function reads the digital, counter, and analog inputs from the SCADAPack 4202 DR I/O. Data are read from 1 digital input and copied to 1 consecutive status registers beginning at startStatusRegister. Data is read from the analog input and copied to 1 input register beginning at startInputRegister. Data are read from the counter inputs and copied to 4 consecutive input registers beginning at startInputRegister + 1.

startStatusRegister is any valid Modbus status register between 10001 and (10000 + NUMSTATUS - 1). startInputRegister is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 4).

The function returns FALSE if startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

Notes

When this function reads data from the transnmitter (controller), it also processes the receiver buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

Digital inputs can also be read with the readCounterInput function.

Counters can also be read with the readCounter function.

Analog inputs can also be read with the readInternalAD function.

See Also

readCounter, readCounterInput

Example

This program displays the values of the 1 digital input, 2 counter inputs and 1 analog input read from SCADAPack 4202 DR I/O.

```
#include "ctools.h"
void main(void)
{
       unsigned reg, counter;
       unsigned long value;

        request_resource(IO_SYSTEM);

        /* Read 4202GFC inputs and write to I/O database */
        ioRead4202Inputs (10001, 30001);

        /* Print digital inputy */
        fprintf(com2, "Register Value");
        fprintf(com2, "\n\r%5u    ", 10001);
        fputc(dbase(MODBUS, 10001) ? '1' :'0', com2);

        /* print analog input */
        reg = 30001;
        fprintf(com2, "\n\r%5u    %d\n\r", reg, dbase(MODBUS,
reg));

        /* print counter inputs */
        fprintf(com2, "Counter  Value\n\r");
        counter = 0;
        for(reg = 30002; reg <= 30005; reg += 2)
        {
                value = (unsigned long) dbase(MODBUS, reg) |
                    ((unsigned long) dbase(MODBUS, reg + 1) <<
16);
   fprintf(com2, "%u:%5u  %lu\n\r", counter++,         reg,
value);
        }

        release_resource(IO_SYSTEM);

       /* Wait here forever */
       while (TRUE)
       {
              NULL;
       }
}
```

ioRead4202DSInputs

Read SCADAPack 4202 DS Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead4202DSInputs(
       unsigned startStatusRegister,
       unsigned startInputRegister
       );
```

Description

The ioRead4202DSInputs function reads the digital, counter, and analog inputs from the SCADAPack 4202 DS I/O. Data are read from 1 digital input and copied to 1 consecutive status registers beginning at startStatusRegister. Data is read from three analog inputs and copied to 3 input register beginning at startInputRegister. Data are read from the counter inputs and copied to 4 consecutive input registers beginning at startInputRegister + 4.

startStatusRegister is any valid Modbus status register between 10001 and (10000 + NUMSTATUS).

startInputRegister is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 6).

The function returns FALSE if startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

Notes

When this function reads data from the SCADAPack 4202 DS I/O it also processes the receiver buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

The digital input can also be read with the readCounterInput function.

Counters can also be read with the readCounter function.

Analog inputs can also be read with the readInternalAD function.

See Also

ioWrite4202DSOutputs, **readCounter**, **readCounterInput**, **readInternalAD**

Example

This program displays the values of the digital input, 2 counter inputs and 3 analog inputs read from the SCADAPack 4202 DS I/O.

```
#include "ctools.h"

void main(void)
{
       unsigned reg, counter;
       unsigned long value;

        request_resource(IO_SYSTEM);

        /* Read 4202 DS inputs and write to I/O database */
        ioRead4202DSInputs (10001, 30001);

        /* Print digital inputy */
        fprintf(com2, "Register Value");
        fprintf(com2, "\n\r%5u    ", 10001);
        fputc(dbase(MODBUS, 10001) ? '1' :'0', com2);

        /* print analog inputs */
        fprintf(com2, "\n\r%5u    %d\n\r", 30001, dbase(MODBUS,
30001));
        fprintf(com2, "%5u    %d\n\r", 30002, dbase(MODBUS,
30002));
        fprintf(com2, "%5u    %d\n\r", 30003, dbase(MODBUS,
30003));

        /* print counter inputs */
        fprintf(com2, "Counter  Value\n\r");
        counter = 0;
        for(reg = 30004; reg <= 30007; reg += 2)
        {
                value = (unsigned long) dbase(MODBUS, reg) |
                    ((unsigned long) dbase(MODBUS, reg + 1) <<
16);
                fprintf(com2, "%u:%5u  %lu\n\r", counter++, reg,
value);
        }

        release_resource(IO_SYSTEM);

       /* Wait here forever */
       while (TRUE)
       {
             release_processor();
       }
}
```

ioRead5505Inputs

Read 5505 Inputs into I/O Database

Syntax

```
#include <ctools.h>
UINT16 ioRead5505Inputs(
       UINT16 moduleAddress,
       UINT16 startStatusRegister,
       UINT16 startInputRegister);
```

Description

The **ioRead5505Inputs** function reads the digital and analog inputs from the 5505 I/O. Data is read from the 16 digital inputs and copied to 16 consecutive status registers beginning at startStatusRegister. Data is read from all 4 analog inputs and copied to 8 consecutive input registers in floating point format beginning at startInputRegister.

The function of the 16 digital inputs is described in the table below.

Point Offset	Function
0	OFF = channel 0 RTD is good ON = channel 0 RTD is open or PWR input is off
1	OFF = channel 0 data in range ON = channel 0 data is out of range
2	OFF = channel 0 RTD is using 3-wire measurement ON = channel 0 RTD is using 4-wire measurement
3	reserved for future use
4	OFF = channel 1 RTD is good ON = channel 1 RTD is open or PWR input is off
5	OFF = channel 1 data in range ON = channel 1 data is out of range
6	OFF = channel 1 RTD is using 3-wire measurement ON = channel 1 RTD is using 4-wire measurement
7	reserved for future use
8	OFF = channel 2 RTD is good ON = channel 2 RTD is open or PWR input is off
9	OFF = channel 2 data in range ON = channel 2 data is out of range
10	OFF = channel 2 RTD is using 3-wire measurement ON = channel 2 RTD is using 4-wire measurement
11	reserved for future use
12	OFF = channel 3 RTD is good

	ON = channel 3 RTD is open or PWR input is off
13	OFF = channel 3 data in range ON = channel 3 data is out of range
14	OFF = channel 3 RTD is using 3-wire measurement ON = channel 3 RTD is using 4-wire measurement
15	reserved for future use

The function returns FALSE if the moduleAddress, startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

moduleAddress is the address of the 5505 module. Valid values are 0 to 15.

startStatusRegister is any valid Modbus status register between 10001 and (10001 + NUMSTATUS - 15).

startInputRegister is any valid Modbus input register between 30001 and (30001 + NUMINPUT - 7).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

This program displays the values of the 16 digital inputs and 4 analog inputs read from 5505 I/O at module address 3.

```
#include <ctools.h>
void main(void)
{
       UINT16 reg;
typedef union
{
       UINT16 intValue[2];
       float  floatValue;
} UF_UNION;
UF_UNION value;

       request_resource(IO_SYSTEM);

       /* Read data from 5505 I/O into I/O database */
       ioRead5505Inputs(3, 10001, 30001);
       /* Print data from I/O database */
       fprintf(com1, "Register          Value");
       for (reg = 10001; reg <= 10016; reg++)
       {
              fprintf(com1, "\n\r%d      ", reg);
              putchar(dbase(MODBUS, reg) ? '1' :'0');
```

```
        }

        for(reg = 30001; reg <= 30008; reg+2)
        {
        value.intValue[1] = dbase(MODBUS, reg); value.intValue[0] =
dbase(MODBUS, reg + 1);
                fprintf(com1, "\n\r%d      %d", reg,
value.floatValue);
        }

        release_resource(IO_SYSTEM);
}
```

ioRead5506Inputs

Read 5506 Inputs into I/O Database

Syntax

```
#include <ctools.h>
UINT16 ioRead5506Inputs(
       UINT16 moduleAddress,
       UINT16 startStatusRegister,
       UINT16 startInputRegister);
```

Description

The **ioRead5506Inputs** function reads the digital and analog inputs from the 5506 I/O. Data is read from the 8 digital inputs and copied to 8 consecutive status registers beginning at startStatusRegister. Data is read from the 8 analog inputs and copied to 8 consecutive input registers beginning at startInputRegister.

The function returns FALSE if the moduleAddress, startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

moduleAddress is the address of the 5506 module. Valid values are 0 to 15.

startStatusRegister is any valid Modbus status register between 10001 and (10001 + NUMSTATUS - 7).

startInputRegister is any valid Modbus input register between 30001 and (30001 + NUMINPUT - 7).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

This program displays the values of the 8 digital inputs and 8 analog inputs read from 5506 I/O at module address 3.

```
#include <ctools.h>

void main(void)
{
       UINT16 reg;

       request_resource(IO_SYSTEM);

       /* Read data from 5506 I/O into I/O database */
       ioRead5506Inputs(3, 10001, 30001);
```

```
        /* Print data from I/O database */
        fprintf(com1, "Register          Value");
        for (reg = 10001; reg <= 10008; reg++)
        {
              fprintf(com1, "\n\r%d      ", reg);
              putchar(dbase(MODBUS, reg) ? '1' :'0');
        }

        for(reg = 30001; reg <= 30008; reg++)
        {
              fprintf(com1, "\n\r%d      %d", reg,
              dbase(MODBUS, reg));
        }

        release_resource(IO_SYSTEM);
}
```

ioRead5601Inputs

Read 5601 Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead5601Inputs(unsigned startStatusRegister, unsigned
startInputRegister);
```

Description

The **ioRead5601Inputs** function reads the digital and analog inputs from a 5601 I/O Module. Data is read from all 16 digital inputs and copied to 16 consecutive status registers beginning at *startStatusRegister*. Data is read from all 8 analog inputs and copied to 8 consecutive input registers beginning at *startInputRegister*.

The function returns FALSE if *startStatusRegister* or *startInputRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startStatusRegister is any valid Modbus status register between 10001 and (10000 + NUMSTATUS - 15). *startInputRegister* is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 7).

Notes

When this function reads data from the 5601 it also processes the receiver buffer for the com3 serial port. If the controller type is a SCADAPack or SCADAPack PLUS, the com3 serial port is also continuously processed automatically.

The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite5601Outputs

Example

This program displays the values of the 16 digital inputs and 8 analog inputs read from a 5601 I/O Module.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Read data from 5601 I/O module and write it
       to I/O database */
       ioRead5601Inputs(10001, 30001);

       /* Print data from I/O database */
       fprintf(com1, "Register          Value");
       for (reg = 10001; reg <= 10016; reg++)
       {
              fprintf(com1, "\n\r%d      ", reg);
              putchar( dbase(MODBUS, reg) ? '1' :'0');
       }

       for(reg = 30001; reg <= 30008; reg++)
       {
              fprintf(com1, "\n\r%d      %d", reg,
              dbase(MODBUS, reg));
       }

       release_resource(IO_SYSTEM);
}
```

ioRead5602Inputs

Read 5602 Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead5602Inputs(unsigned startStatusRegister, unsigned
startInputRegister);
```

Description

The **ioRead5602Inputs** function reads the inputs from a 5602 I/O Module as digital or analog inputs. Data is read from the 5 analog inputs and copied to 5 consecutive input registers beginning at *startInputRegister*. The same 5 analog inputs are also read as 5 digital inputs and copied to 5 consecutive status registers beginning at *startStatusRegister*.

A digital input is ON if the corresponding filtered analog input value is greater than or equal to 20% of its full scale value, otherwise it is OFF. Analog input 0 to 4 correspond to digital inputs 0 to 4.

The function returns FALSE if *startStatusRegister* or *startInputRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startStatusRegister is any valid Modbus status register between 10001 and (10000 + NUMSTATUS - 4). *startInputRegister* is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 4).

Notes

When this function reads data from the 5602 it also processes the receiver buffer for the com4 serial port. If the controller type is a SCADAPack LIGHT or SCADAPack PLUS, the com4 serial port is also continuously processed automatically.

The additional service to the com4 receiver caused by this function does not affect the normal automatic operation of com4.

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite5602Outputs

Example

This program displays the values of the 5 inputs read from a 5602 I/O Module as both digital and analog inputs.

```
#include <ctools.h>

void main(void)
{
	unsigned reg;

	request_resource(IO_SYSTEM);

	/* Read data from 5602 I/O module and write it
	to I/O database */
	ioRead5602Inputs(10001, 30001);

	/* Print data from I/O database */
	fprintf(com1, "Register          Value");
	for (reg = 10001; reg <= 10005; reg++)
	{
		fprintf(com1, "\n\r%d      ", reg);
		putchar( dbase(MODBUS, reg) ? '1' :'0');
	}

	for(reg = 30001; reg <= 30005; reg++)
	{
		fprintf(com1, "\n\r%d      %d", reg,
		dbase(MODBUS, reg));
	}
	release_resource(IO_SYSTEM);
}
```

ioRead5604Inputs

Read 5604 Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead5604Inputs(
       unsigned startStatusRegister,
       unsigned startInputRegister);
```

Description

The ioRead5604Inputs function reads the digital and analog inputs from the 5604 I/O. Data is read from the 35 digital inputs and copied to 35 consecutive status registers beginning at startStatusRegister. Data is read from the 10 analog inputs and copied to 10 consecutive input registers beginning at startInputRegister.

The function returns FALSE if startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startStatusRegister is any valid Modbus status register between 10001 and (10001 + NUMSTATUS - 35).

startInputRegister is any valid Modbus input register between 30001 and (30001 + NUMINPUT - 10).

Notes

When this function reads data from the 5604 I/O it also processes the receiver buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite5604Outputs

Example

This program displays the values of the 35 digital inputs and 10 analog inputs read from 5604 I/O.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;
```

```
        request_resource(IO_SYSTEM);

        /* Read data from 5604 I/O into I/O database */
        ioRead5604Inputs(10001, 30001);

        /* Print data from I/O database */
        fprintf(com1, "Register          Value");
        for (reg = 10001; reg <= 10035; reg++)
        {
              fprintf(com1, "\n\r%d      ", reg);
              putchar(dbase(MODBUS, reg) ? '1' :'0');
        }

        for(reg = 30001; reg <= 30010; reg++)
        {
              fprintf(com1, "\n\r%d      %d", reg,
              dbase(MODBUS, reg));
        }

        release_resource(IO_SYSTEM);
}
```

ioRead5606Inputs

Read 5606 Inputs into I/O Database

Syntax

```
#include <ctools.h>
UINT16 ioRead5606Inputs(
       UINT16 moduleAddress,
       UINT16 startStatusRegister,
       UINT16 startInputRegister);
```

Description

The **ioRead5606Inputs** function reads the digital and analog inputs from the 5606 I/O. Data is read from the 40 digital inputs and copied to 40 consecutive status registers beginning at startStatusRegister. Data is read from all 8 analog inputs and copied to 8 consecutive input registers beginning at startInputRegister.

The function returns FALSE if the moduleAddress, startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

moduleAddress is the address of the 5606 module. Valid values are 0 to 7.

startStatusRegister is any valid Modbus status register between 10001 and (10001 + NUMSTATUS - 39).

startInputRegister is any valid Modbus input register between 30001 and (30001 + NUMINPUT - 7).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

This program displays the values of the 40 digital inputs and 8 analog inputs read from 5606 I/O at module address 3.

```
#include <ctools.h>

void main(void)
{
       UINT16 reg;

       request_resource(IO_SYSTEM);

       /* Read data from 5606 I/O into I/O database */
       ioRead5606Inputs(3, 10001, 30001);
```

```
        /* Print data from I/O database */
        fprintf(com1, "Register          Value");
        for (reg = 10001; reg <= 10040; reg++)
        {
              fprintf(com1, "\n\r%d      ", reg);
              putchar(dbase(MODBUS, reg) ? '1' :'0');
        }

        for(reg = 30001; reg <= 30008; reg++)
        {
              fprintf(com1, "\n\r%d      %d", reg,
              dbase(MODBUS, reg));
        }

        release_resource(IO_SYSTEM);
}
```

ioRead8Ain

Read 8 Analog Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead8Ain(unsigned moduleAddress, unsigned
startInputRegister);
```

Description

The **ioRead8Ain** function reads any 8 point Analog Input Module at the specified *moduleAddress*. Data is read from all 8 analog inputs and copied to 8 consecutive input registers beginning at *startInputRegister*.

The function returns FALSE if the *moduleAddress* or *startInputRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startInputRegister* is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 7).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead4Ain

Example

This program displays the values of the 8 analog inputs read from an 8 point Analog Input Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Read data from digital input module and     write it to
I/O database */
       ioRead8Ain(0, 30001);
```

```
        /* Print data from I/O database */
        fprintf(com1, "Register          Value\n\r");
        for(reg = 30001; reg <= 30008; reg++)
        {
               fprintf(com1, "%d   %d\n\r", reg,
        dbase(MODBUS, reg));
        }

        release_resource(IO_SYSTEM);
}
```

ioRead8Din

Read 8 Digital Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioRead8Din(unsigned moduleAddress, unsigned
startStatusRegister);
```

Description

The **ioRead8Din** function reads any 8 point Digital Input Module at the specified *moduleAddress*. Data is read from the 8 digital inputs and copied to 8 consecutive status registers beginning at *startStatusRegister*.

The function returns FALSE if the *moduleAddress* or *startStatusRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startStatusRegister* is any valid Modbus status register between 10001 and (10000 + NUMSTATUS - 7).

Notes

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead16Din

Example

This program displays the values of the 8 digital inputs read from an 8 point Digital Input Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Read data from digital input module and     write it to
I/O database */
       ioRead8Din(0, 10001);
```

```
        /* For each digital input on the module */
        fprintf(com1, "Register          Value");
        for (reg = 10001; reg <= 10008; reg++)
        {
              fprintf(com1, "\n\r%d      ", reg);

              putchar( dbase(MODBUS, reg) ? '1' :'0');
        }

        release_resource(IO_SYSTEM);
}
```

ioReadLPInputs

Read SCADAPack LP Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioReadLPInputs (unsigned startStatusRegister, unsigned
startInputRegister);
```

Description

The ioReadLPInputs function reads the digital and analog inputs from the SCADAPack LP I/O. Data is read from the 16 digital inputs and copied to 16 consecutive status registers beginning at startStatusRegister. Data is read from the 8 analog inputs and copied to 8 consecutive input registers beginning at startInputRegister.

The function returns FALSE if startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startStatusRegister is any valid Modbus status register between 10001 and (10000 + NUMSTATUS - 15). startInputRegister is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 7).

Notes

When this function reads data from the SCADAPack LP I/O it also processes the receiver buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

Data is not copied to the I/O database for registers that are currently forced.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

This program displays the values of the 16 digital inputs and 8 analog inputs read from SCADAPack LP I/O.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Read data from LP I/O and write it to I/O database */
       ioReadLPInputs (10001, 30001);
```

```
        /* Print data from I/O database */
        fprintf(com1, "Register           Value");
        for (reg = 10001; reg <= 10016; reg++)
        {
              fprintf(com1, "\n\r%d      ", reg);
              putchar( dbase(MODBUS, reg) ? '1' :'0');
        }

        for(reg = 30001; reg <= 30008; reg++)
        {
              fprintf(com1, "\n\r%d      %d", reg,
              dbase(MODBUS, reg));
        }

        release_resource(IO_SYSTEM);
}
```

ioReadSP100Inputs

Read SCADAPack 100 Inputs into I/O Database

Syntax

```
#include <ctools.h>
unsigned ioReadSP100Inputs(unsigned startStatusRegister, unsigned
startInputRegister);
```

Description

The ioReadSP100Inputs function reads the digital and analog inputs from the SCADAPack 100 I/O. Data is read from the 6 digital inputs and copied to 6 consecutive status registers beginning at startStatusRegister. Data is read from the 6 analog inputs and one counter input, and copied to 8 consecutive input registers beginning at startInputRegister.

The function returns FALSE if startStatusRegister or startInputRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startStatusRegister is any valid Modbus status register between 10001 and (10000 + NUMSTATUS - 5). startInputRegister is any valid Modbus input register between 30001 and (30000 + NUMINPUT - 7).

Notes

Data is not copied to the I/O database for registers that are currently forced.

Data from the four external analog inputs is copied to the first four input registers.

Data from the temperature sensor is copied to the fifth input register.

Data from the battery voltage sensor is copied to the sixth input register.

Data from the counter input is copied to the seventh and eighth input registers.

To read data from an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWriteSP100Outputs

Example

This program displays the values of the 6 digital inputs, 6 analog inputs, and the counter input read from SCADAPack 100 I/O.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;
       unsigned long count;
```

```
        request_resource(IO_SYSTEM);

        /* Read data from I/O and write it to I/O database */
        ioReadSP100Inputs(10001, 30001);

        /* Print digital data from I/O database */
        for (reg = 10001; reg <= 10006; reg++)
        {
               fprintf(com1, "Register %d = %d\r\n", reg,
               dbase(MODBUS, reg));
        }
        fprintf(com1, "\r\n");

        /* Print analog data from I/O database */
        for(reg = 30001; reg <= 30006; reg++)
        {
               fprintf(com1, "Regsiter %d = %d\n\r", reg,
               dbase(MODBUS, reg));
        }
        fprintf(com1, "\r\n");

        /* Print counter data from I/O database */
        count = dbase(MODBUS, 30006);
        count += ((unsigned long) dbase(MODBUS, reg)) << 16;

        fprintf(com1, "Registers 30006 & 30007 = %ul\r\n", reg,
        count);

        release_resource(IO_SYSTEM);
}
```

ioRefresh

Update Outputs with Internal Data

Syntax

```
#include <ctools.h>
void ioRefresh(void);
```

Description

The **ioRefresh** function resets devices on the 5000 I/O bus. Input channels are scanned to update their values from the I/O hardware. Output channels are scanned to write their values from output tables in memory.

Notes

This function is normally only used by the sleep function to restore output states when the controller wakes.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioClear, ioReset

ioReset

Reset 5000 I/O Modules

Syntax

```
#include <ctools.h>
void ioReset(unsigned state);
```

Description

The **ioReset** function sets the state of the 5000 I/O bus reset signal. *state* may be TRUE or FALSE.

The reset signal restarts all devices on the 5000 I/O bus. Output modules clear all their output points. Input modules restart their input scanning. All modules remain in the reset state until the reset signal is set to FALSE.

Notes

Do not leave the reset signal in the TRUE state. This will disable I/O.

The **ioClear** function provides a more effective method of resetting the I/O system.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioClear

ioWrite16Dout

Write to 16 Digital Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite16Dout(unsigned moduleAddress, unsigned
startCoilRegister);
```

Description

The **ioWrite16Dout** function writes data to any 16 point Digital Output Module at the specified *moduleAddress*. Data is read from 16 consecutive coil registers beginning at *startCoilRegister*, and written to the 16 digital outputs.

The function returns FALSE if the *moduleAddress* or *startCoilRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startCoilRegister* is any valid Modbus coil register between 00001 and (NUMCOIL - 15).

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment. Refer to the section **Overview of Functions** for details.

This function is contained in the ctools.lib library. Load this library in you linker command (.cmd) file as shown in the sample file appram.cmd in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite8Dout

Example

This program turns ON all 16 digital outputs of a 16 point Digital Output Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Write data to I/O database */
       for (reg = 1; reg <= 16; reg++)
       {
              setdbase(MODBUS, reg, 1);
```

```
        }

        /* Write data from I/O database to digital
        output module */
        ioWrite16Dout(0, 1);

        release_resource(IO_SYSTEM);
}
```

ioWrite32Dout

Write to 32 Digital Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite32Dout(
       UINT16 moduleAddress,
       UINT16 startCoilRegister);
```

Description

The ioWrite32Dout function writes data to any 32-point Digital Output Module at the specified moduleAddress. Data is read from 32 consecutive coil registers beginning at startCoilRegister, and written to the 32 digital outputs.

The function returns FALSE if the moduleAddress or startCoilRegister is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for moduleAddress is 0 to 15. startCoilRegister is any valid Modbus coil register between 00001 and (NUMCOIL - 31).

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment.

This function is contained in the ctools.lib library. Load this library in you linker command (.cmd) file as shown in the sample file appram.cmd in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite8Dout, **ioWrite16Dout**

Example

This program turns ON all 32 digital outputs of a 32 point Digital Output Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Write data to I/O database */
       for (reg = 1; reg <= 32; reg++)
       {
              setdbase(MODBUS, reg, 1);
```

```
        }

        /* Write data from I/O database to digital
        output module */
        ioWrite32Dout(0, 1);

        release_resource(IO_SYSTEM);
}
```

ioWrite8Dout

Write to 8 Digital Outputs from I/O Database

Syntax

```
#include <iomodule.h>
unsigned ioWrite8Dout(unsigned moduleAddress, unsigned
startCoilRegister);
```

Description

The **ioWrite8Dout** function writes data to any 8 point Digital Output Module at the specified *moduleAddress*. Data is read from 8 consecutive coil registers beginning at *startCoilRegister*, and written to the 8 digital outputs.

The function returns FALSE if the *moduleAddress* or *startCoilRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startCoilRegister* is any valid Modbus coil register between 00001 and (NUMCOIL - 7).

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment. Refer to the section **Overview of Functions** for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite16Dout

Example

This program turns ON all 8 digital outputs of an 8 point Digital Output Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Write data to I/O database */
       for (reg = 1; reg <= 8; reg++)
       {
              setdbase(MODBUS, reg, 1);
       }
```

```
        /* Write data from I/O database to digital
        output module */
        ioWrite8Dout(0, 1);

        release_resource(IO_SYSTEM);
}
```

ioWrite2Aout

Write to 2 Analog Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite2Aout(unsigned moduleAddress, unsigned
startHoldingRegister);
```

Description

The **ioWrite2Aout** function writes data to any 2 point Analog Output Module at the specified *moduleAddress*. Data is read from 2 consecutive holding registers beginning at *startHoldingRegister*, and written to the 2 analog outputs.

The function returns FALSE if the *moduleAddress* or *startHoldingRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startHoldingRegister* is any valid Modbus holding register between 40001 and (40000 + NUMHOLDING - 1).

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment. Refer to the section **Overview of Functions** for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite4Aout, ioWrite5303Aout

Example

This program sets both analog outputs to half scale on a 2 point Analog Output Module at module address 0.

```
#include <ctools.h>

void main(void)
{
       request_resource(IO_SYSTEM);

       /* Write data to I/O database */
       setdbase(MODBUS, 40001, 16384);
       setdbase(MODBUS, 40002, 16384);

       /* Write data from I/O database to analog
       output module */
       ioWrite2Aout(0, 40001);
```

```
        release_resource(IO_SYSTEM);
}
```

ioWrite4Aout

Write to 4 Analog Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite4Aout(unsigned moduleAddress, unsigned
startHoldingRegister);
```

Description

The **ioWrite4Aout** function writes data to any 4 point Analog Output Module at the specified *moduleAddress*. Data is read from 4 consecutive holding registers beginning at *startHoldingRegister*, and written to the 4 analog outputs.

The function returns FALSE if the *moduleAddress* or *startHoldingRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

The valid range for *moduleAddress* is 0 to 15. *startHoldingRegister* is any valid Modbus holding register between 40001 and (40000 + NUMHOLDING - 3).

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment. Refer to the section **Overview of Functions** for details.

This function is contained in the ctools.lib library. Load this library in you linker command (.cmd) file as shown in the sample file appram.cmd in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite2Aout, ioWrite5303Aout

Example

This program sets all 4 analog outputs to half scale on a 4 point Analog Output Module at module address 0.

```
#include <ctools.h>

void main(void)
{
	unsigned reg;

	request_resource(IO_SYSTEM);

	/* Write data to I/O database */
	for (reg = 40001; reg <= 40004; reg++)
	{
		setdbase(MODBUS, reg, 16384);
	}
```

```
        /* Write data from I/O database to analog
        output module */
        ioWrite4Aout(0, 40001);

        release_resource(IO_SYSTEM);
}
```

ioWrite4AoutChecksum

Write to 4 Point Analog Output Module with Checksum

Syntax

```
#include <ctools.h>
UINT16 ioWrite4AoutChecksum(
       UINT16 moduleAddress,
       UINT16 startHoldingRegister
       )
```

Description

The ioWrite4AoutChecksum function writes data to a 4-point analog output module with checksum support. Output data comes from the I/O database. The function can be used with 5304 analog output modules. Use the isaWrite4Aout function for all other analog output modules.

The function has two parameters.

- moduleAddress is the address of the module. The valid range is 0 to 15.
- Data are read from 4 consecutive holding registers and written to 4 analog outputs. startHoldingRegister is any valid Modbus holding register between 40001 and (40001 + NUMHOLDING - 4).

The function returns FALSE if the moduleAddress or startHoldingRegister is invalid, or if an I/O error occurs; otherwise TRUE is returned.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

This function is contained in the ctools.lib library. Load this library in you linker command (.cmd) file as shown in the sample file appram.cmd in your ctools directory.

To write data to an I/O Module continuously, add the module to the Register Assignment.

See Also

ioWrite2Aout, **ioWrite4Aout**, **ioWrite5303Aout**

Example

This program sets all 4 analog outputs to half scale on a 5304 Analog Output Module at module at address 0.

```
#include <ctools.h>

void main(void)
{
       UINT16 reg;
       request_resource(IO_SYSTEM);
```

```
        /* Write data to I/O database */
        for (reg = 40001; reg <= 40004; reg++)
        {
               setdbase(MODBUS, reg, 16384);
        }
        /* Write I/O database to 5304 analog output module */
        ioWrite4AoutChecksum(0, 40001);

        release_resource(IO_SYSTEM);
}
```

ioWrite4202Outputs

Write to SCADAPack 4202 DR Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite4202Outputs(
       unsigned startHoldingRegister
       );
```

Description

The ioWrite4202Outputs function writes data to the analog output of the SCADAPack 4202 DR I/O. Analog data is read from 1 holding register beginning at startHoldingRegister and written to the analog output.

startHoldingRegister is any valid Modbus holding register between 40001 and (4000 + NUMHOLDING).

The function returns FALSE if startHoldingRegister is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

Notes

When this function writes data to the SCADAPack 4202 DR I/O it also processes the transmit buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead4202Inputs, ioWrite4202OutputsEx

Example

This program sets the analog output to full scale.

```
#include <ctools.h>

void main(void)
{
       request_resource(IO_SYSTEM);

       /* Write analog data to I/O database */
       setdbase(MODBUS, 40001, 32767);

       /* Write data from I/O database to 4202 DR output */
       ioWrite4202Outputs(40001);
```

```
        release_resource(IO_SYSTEM);
}
```

ioWrite4202OutputsEx

Write to SCADAPack 4202 DR with Extended Outputs, from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite4202OutputsEx(
       unsigned startCoilRegister,
       unsigned startHoldingRegister
       );
```

Description

The ioWrite4202OutputsEx function writes data to the outputs of the SCADAPack 4202 DR with Extended I/O (digital output). Digital data is read from one coil register starting at startCoilRegister and written to the digital output. Analog data is read from 1 holding register beginning at startHoldingRegister and written to the analog output.

startCoilRegister is any valid Modbus coil register between 1 and (NUMCOIL).

startHoldingRegister is any valid Modbus holding register between 40001 and (4000 + NUMHOLDING).

The function returns FALSE if startCoilRegister or startHoldingRegister are invalid, or if an I/O error has occurred; otherwise TRUE is returned.

Notes

When this function writes data to the SCADAPack 4202 DR I/O it also processes the transmit buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead4202Inputs

Example

This program sets the analog output to full scale and turns on the digital output.

```
#include <ctools.h>
void main(void)
{
       request_resource(IO_SYSTEM);

       /* Write output data to I/O database */
```

```
        setdbase(MODBUS, 1, 1);
        setdbase(MODBUS, 40001, 32767);

        /* Write data from I/O database to 4202 DR outputs */
        ioWrite4202OutputsEx(1, 40001);

        release_resource(IO_SYSTEM);
}
```

ioWrite4202DSOutputs

Write to SCADAPack 4202 DS Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite4202DSOutputs(
       unsigned startCoilRegister
       );
```

Description

The ioWrite4202DSOutputs function writes data to the outputs of the SCADAPack 4202 DS I/O module. Digital data is read from two coil registers starting at startCoilRegister and written to the digital outputs.

startCoilRegister is any valid Modbus coil register between 1 and (NUMCOIL - 1).

The function returns FALSE if startCoilRegister is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

Notes

When this function writes data to the SCADAPack 4202 DS I/O it also processes the transmit buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead4202DSInputs

Example

This program turns on the digital outputs.

```
#include <ctools.h>

void main(void)
{
       request_resource(IO_SYSTEM);

       /* Write output data to I/O database */
       setdbase(MODBUS, 1, 1);
       setdbase(MODBUS, 2, 1);

       /* Write data from I/O database to 4202 DS outputs */
       ioWrite4202DSOutputs(1);
```

```
        release_resource(IO_SYSTEM);
}
```

ioWrite5303Aout

Write to 5303 Analog Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite5303Aout(unsigned startHoldingRegister);
```

Description

The **ioWrite5303Aout** function writes data to the 2 points on a 5303 SCADAPack Analog Output Module. Data is read from 2 consecutive holding registers beginning at *startHoldingRegister*, and written to the 2 analog outputs.

The function returns FALSE if *startHoldingRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startHoldingRegister is any valid Modbus holding register between 40001 and (40000 + NUMHOLDING - 1).

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment. Refer to the section *I/O Database and Register Assignment* for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioWrite2Aout, ioWrite5303Aout

Example

This program sets both analog outputs to half scale on a 5303 Analog Output Module.

```
#include <ctools.h>

void main(void)
{
       request_resource(IO_SYSTEM);

       /* Write data to I/O database */
       setdbase(MODBUS, 40001, 16384);
       setdbase(MODBUS, 40002, 16384);

       /* Write data from I/O database to analog
       output module */
       ioWrite5303Aout(40001);
```

```
        release_resource(IO_SYSTEM);
}
```

ioWrite5505Outputs

Write to 5505 Configuration from I/O Database

Syntax

```
#include <ctools.h>
UINT16 ioWrite5505Outputs(
       UINT16 moduleAddress,
       UINT16 inputType[4],
       UINT16 inputFilter
);
```

Description

The ioWrite5505Outputs function writes configuration data to the 5505 I/O module.

The function returns FALSE if moduleAddress is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

moduleAddress is the address of the 5505 module. Valid values are 0 to 15.

inputType is an array of 4 values indicating the input range for the corresponding analog input. Valid values are

- 0 = RTD in deg Celsius
- 1 = RTD in deg Fahrenheit
- 2 = RTD in deg Kelvin
- 3 = resistance measurement in ohms.

inputFilter is the analog input filter setting. Valid values are.

- 0 = 0.5 s
- 1 = 1 s
- 2 = 2 s
- 3 = 4 s

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

This program writes configuration data to a 5505 I/O module at module address 5.

```
#include <ctools.h>
```

```
void main(void)
{
       UINT16 index;
       UINT16 inputType[4];
       UINT16 inputFilter;

       request_resource(IO_SYSTEM);

       /* set the input types */
       for (index = 0; index < 4; index++)
       {
              inputType[index] = 1; // RTD in deg F
       }

       /* set filter */
       inputFilter = 3;           // maximum filter

       /* Write configuration data to 5505 I/O module */
       ioWrite5505Outputs(5, inputType, inputFilter);

       release_resource(IO_SYSTEM);
}
```

ioWrite5506Outputs

Write to 5506 Configuration from I/O Database

Syntax

```
#include <ctools.h>
UINT16 ioWrite5506Outputs(
        UINT16 moduleAddress,
        UINT16 inputType[8],
        UINT16 inputFilter,
        UINT16 scanFrequency
);
```

Description

The **ioWrite5506Outputs** function writes configuration data to the 5506 I/O module.

The function returns FALSE if moduleAddress is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

moduleAddress is the address of the 5506 module. Valid values are 0 to 15.

inputType is an array of 8 values indicating the input range for the corresponding analog input. Valid values are

- 0 = 0 to 5 V
- 1 = 1 to 5 V
- 2 = 0 to 20 mA
- 3 = 4 to 20 mA.

inputFilter is the analog input filter setting. Valid values are.

- 0 = 3 Hz
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz

scanFrequency is the scan frequency setting. Valid values are.

- 0 = 60 Hz
- 1 = 50 Hz

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead5606Inputs

Example

This program writes configuration data to a 5506 I/O module at module address 5.

```
#include <ctools.h>

void main(void)
{
	UINT16 index;
	UINT16 inputType[8];
	UINT16 inputFilter;
	UINT16 scanFrequency;

	request_resource(IO_SYSTEM);

	/* set the input types */
	for (index = 0; index < 8; index++)
	{
		inputType[index] = 1; // 1 to 5 V
	}

	/* set filter and frequency */
	inputFilter = 3;		// minimum filter
	scanFrequency = 0;	// 60 Hz

	/* Write configuration data to 5506 I/O module */
	ioWrite5506Outputs(5, inputType, inputFilter,
scanFrequency);

	release_resource(IO_SYSTEM);
}
```

ioWrite5601Outputs

Write to 5601 Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite5601Outputs(unsigned startCoilRegister);
```

Description

The **ioWrite5601Outputs** function writes data to the digital outputs of a 5601 I/O Module. Data is read from 12 consecutive coil registers beginning at *startCoilRegister*, and written to the 12 digital outputs.

The function returns FALSE if *startCoilRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startCoilRegister is any valid Modbus coil register between 00001 and (NUMCOIL - 11).

Notes

When this function writes data to the 5601 it also services to the transmit buffer of the com3 serial port. If the controller type is a SCADAPack or SCADAPack PLUS, the com3 serial port is also continuously processed automatically.

The additional service to the com3 transmitter caused by this function does not affect the normal automatic operation of com3.

To write data to an I/O Module continuously, add the module to the Register Assignment. Refer to the section **Overview of Functions** for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead5601Inputs

Example

This program turns ON all 12 digital outputs of a 5601 I/O Module.

```
#include <ctools.h>

void main(void)
{
        unsigned reg;

        request_resource(IO_SYSTEM);

        /* Write data to I/O database */
```

```
        for (reg = 1; reg <= 12; reg++)
        {
                setdbase(MODBUS, reg, 1);
        }

        /* Write data from I/O database to 5601 */
        ioWrite5601Outputs(1);

        release_resource(IO_SYSTEM);
}
```

ioWrite5602Outputs

Write to 5602 Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite5602Outputs(unsigned startCoilRegister);
```

Description

The **ioWrite5602Outputs** function writes data to the digital outputs of a 5602 I/O Module. Data is read from 2 consecutive coil registers beginning at *startCoilRegister*, and written to the 2 digital outputs.

The function returns FALSE if *startCoilRegister* is invalid or if an I/O error has occurred; otherwise TRUE is returned.

startCoilRegister is any valid Modbus coil register between 00001 and (NUMCOIL - 1).

Notes

When this function writes data to the 5602 it also services to the transmit buffer of the com4 serial port. If the controller type is a SCADAPack LIGHT or SCADAPack PLUS, the com4 serial port is also continuously processed automatically.

The additional service to the com4 transmitter caused by this function does not affect the normal automatic operation of com4.

To write data to an I/O Module continuously, add the module to the Register Assignment. Refer to the section **Overview of Functions** for details.

This function is contained in the *ctools.lib* library. Load this library in you linker command (.cmd) file as shown in the sample file *appram.cmd* in your ctools directory.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead5602Inputs

Example

This program turns ON both digital outputs of a 5602 I/O Module.

```
#include <ctools.h>
void main(void)
{
        unsigned reg;
        request_resource(IO_SYSTEM);

        /* Write data to I/O database */
        setdbase(MODBUS, 1, 1);
```

```
        setdbase(MODBUS, 2, 1);

        /* Write data from I/O database to 5602 */
        ioWrite5602Outputs(1);

        release_resource(IO_SYSTEM);
}
```

ioWrite5604Outputs

Write to 5604 Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWrite5604Outputs(
       unsigned startCoilRegister,
       unsigned startHoldingRegister);
```

Description

The ioWrite5604Outputs function writes data to the digital and analog outputs of the 5604 I/O. Digital data is read from 36 consecutive coil registers beginning at startCoilRegister, and written to the 36 digital outputs. Analog data is read from 2 consecutive holding registers beginning at startHoldingRegister and written to the 2 analog outputs.

The function returns FALSE if startCoilRegister is invalid, if startHoldingRegister is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

startCoilRegister is any valid Modbus coil register between 00001 and (1 + NUMCOIL - 36).

startHoldingRegister is any valid Modbus holding register between 40001 and (40001 + NUMHOLDING - 2).

Notes

When this function writes data to the 5604 I/O it also processes the transmit buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 transmitter caused by this function does not affect the normal automatic operation of com3.

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead5604Inputs

Example

This program turns on all 32 external digital outputs and sets the analog outputs to full scale. The internal digital outputs are turned off.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;
```

```
        request_resource(IO_SYSTEM);

        /* Write digital data to I/O database */
        for (reg = 1; reg <= 32; reg++)
        {
               setdbase(MODBUS, reg, 1);
        }

        for (reg = 33; reg <= 36; reg++)
        {
               setdbase(MODBUS, reg, 0);
        }

        /* Write analog data to I/O database */
        for (reg = 40001; reg <= 40002; reg++)
        {
               setdbase(MODBUS, reg, 32767);
        }

        /* Write data from I/O database to 5604 I/O */
        ioWrite5604Outputs(1, 40001);

        release_resource(IO_SYSTEM);
}
```

ioWrite5606Outputs

Write to 5606 Outputs from I/O Database

Syntax

```
#include <ctools.h>
UINT16 ioWrite5606Outputs(
       UINT16 moduleAddress,
       UINT16 startCoilRegister,
       UINT16 startHoldingRegister,
       UINT16 inputType[8],
       UINT16 inputFilter,
       UINT16 scanFrequency,
       UINT16 outputType
);
```

Description

The **ioWrite5606Outputs** function writes data to the digital and analog outputs of the 5606 I/O. Digital data is read from 16 consecutive coil registers beginning at startCoilRegister, and written to the 16 digital outputs. Analog data is read from 2 consecutive holding registers beginning at startHoldingRegister and written to the 2 analog outputs.

The function returns FALSE if moduleAddress, startCoilRegister or startHoldingRegister is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

moduleAddress is the address of the 5606 module. Valid values are 0 to 7.

startCoilRegister is any valid Modbus coil register between 00001 and (1 + NUMCOIL - 15).

startHoldingRegister is any valid Modbus holding register between 40001 and (40001 + NUMHOLDING - 1).

inputType is an array of 8 values indicating the input range for the corresponding analog input. Valid values are

- 0 = 0 to 5V
- 1 = 0 to 10 V
- 2 = 0 to 20 mA
- 3 = 4 to 20 mA.

inputFilter is the analog input filter setting. Valid values are.

- 0 = 3 Hz
- 1 = 6 Hz
- 2 = 11 Hz
- 3 = 30 Hz

scanFrequency is the scan frequency setting. Valid values are.

- 0 = 60 Hz
- 1 = 50 Hz

outputType selects the type of analog outputs on the module. Valid values are

- 0 = 0 to 20 mA
- 1 = 4 to 20 mA.

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioRead5606Inputs

Example

This program turns on all 16 external digital outputs and sets the analog outputs to full scale. The internal digital outputs are turned off. The module address is 5.

```
#include <ctools.h>

void main(void)
{
       UINT16 index;
       UINT16 inputType[8];
       UINT16 inputFilter;
       UINT16 scanFrequency;
       UINT16 outputType;

       request_resource(IO_SYSTEM);

       /* Write digital data to I/O database */
       for (index = 1; index <= 16; index ++)
       {
              setdbase(MODBUS, index, 1);
       }

       /* Write analog data to I/O database */
       for (index = 40001; index <= 40002; index ++)
       {
              setdbase(MODBUS, index, 32767);
       }

       /* set the input types */
       for (index = 0; index < 8; index++)
       {
              inputType[index] = 1; // 0 to 10 V
       }
```

```
        /* set filter and frequency */
        inputFilter = 3;            // minimum filter
        scanFrequency = 0;  // 60 Hz

        /* set analog output type to 4-20 mA */
        outputType = 1;

        /* Write data from I/O database to 5606 I/O */
        ioWrite5606Outputs(5, 1, 40001, inputType, inputFilter,
scanFrequency, outputType);

        release_resource(IO_SYSTEM);
}
```

ioWriteLPOutputs

Write to SCADAPack LP Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWriteLPOutputs (unsigned startCoilRegister, unsigned
startHoldingRegister);
```

Description

The ioWriteLPOutputsfunction writes data to the digital and analog outputs of the SCADAPack LP I/O. Digital data is read from 12 consecutive coil registers beginning at startCoilRegister, and written to the 12 digital outputs. Analog data is read from 2 consecutive holding registers beginning at startHoldingRegister and written to the 2 analog outputs.

The function returns FALSE if startCoilRegister is invalid, if startHoldingRegister is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

startCoilRegister is any valid Modbus coil register between 00001 and (NUMCOIL - 11).

startHoldingRegister is any valid Modbus holding register between 40001 and (NUMHOLDING - 2).

Notes

When this function writes data to the SCADAPack LP I/O it also processes the transmit buffer for the com3 serial port. The com3 serial port is also continuously processed automatically. The additional service to the com3 receiver caused by this function does not affect the normal automatic operation of com3.

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioReadLPInputs

Example

This program turns on all 12 digital outputs and sets the analog outputs to full scale.

```
#include <ctools.h>

void main(void)
{
        unsigned reg;

        request_resource(IO_SYSTEM);
```

```
        /* Write digital data to I/O database */
        for (reg = 1; reg <= 12; reg++)
        {
              setdbase(MODBUS, reg, 1);
        }

        /* Write analog data to I/O database */
        for (reg = 40001; reg <= 40002; reg++)
        {
              setdbase(MODBUS, reg, 32767);
        }

        /* Write data from I/O database to SCADAPack LP I/O */
        ioWriteLPOutputs (1, 40001);

        release_resource(IO_SYSTEM);
}
```

ioWriteSP100Outputs

Write to SCADAPack 100 Outputs from I/O Database

Syntax

```
#include <ctools.h>
unsigned ioWriteSP100Outputs(unsigned startCoilRegister);
```

Description

The ioWriteSP100Outputs function writes data to the digital outputs of the SCADAPack 100 I/O. Digital data is read from 6 consecutive coil registers beginning at startCoilRegister, and written to the 6 digital outputs.

The function returns FALSE if startCoilRegister is invalid, or if an I/O error has occurred; otherwise TRUE is returned.

startCoilRegister is any valid Modbus coil register between 00001 and (NUMCOIL - 5).

Notes

To write data to an I/O Module continuously, add the module to the Register Assignment.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

ioReadSP100Inputs

Example

This program turns on all 6 digital outputs.

```
#include <ctools.h>

void main(void)
{
       unsigned reg;

       request_resource(IO_SYSTEM);

       /* Write digital data to I/O database */
       for (reg = 1; reg <= 6; reg++)
       {
              setdbase(MODBUS, reg, 1);
       }

       /* Write data from I/O database to SCADAPack 100 I/O */
       ioWriteSP100Outputs(1);

       release_resource(IO_SYSTEM);
}
```

jiffy

Read System Clock

Syntax

```
#include <ctools.h>
unsigned long jiffy(void);
```

Description

The **jiffy** function returns the current value of the system jiffy clock. The jiffy clock increments every 1/60 second. The jiffy clock rolls over to 0 after 5183999. This is the number of 1/60 second intervals in a day.

Notes

The real time clock and the jiffy clock are not related. They may drift slightly with respect to each other over several days.

Use the jiffy clock to measure times with resolution better than the 1/10th resolution provided by timers.

See Also

interval, setjiffy

Example

This program uses the jiffy timer to determine the execution time of a section of code. The section is run 10 times to provide a longer time base for the measurement.

```
#include <ctools.h>
void main(void)
{
       int iterations = 10;
       int i;

       setjiffy(0UL);
       for(i=0; i<=iterations; i++)
       {
              /* statements to time */
       }
       printf("average time=%ld jiffies",
              jiffy()/iterations);
}
```

ledGetDefault

Read LED Power Control Parameters

Syntax

```
#include <ctools.h>
struct ledControl_tag ledGetDefault(void);
```

Description

The **ledGetDefault** routine returns the default LED power control parameters. The controller controls LED power to 5000 I/O modules. To conserve power, the LEDs can be disabled.

The user can change the LED power setting with the LED POWER switch on the controller. The LED power returns to its default state after a user specified time period.

Example

See the example for the **ledSetDefault** function.

ledPower

Set LED Power State

Syntax

```
#include <ctools.h>
unsigned ledPower(unsigned state);
```

Description

The **ledPower** function sets the LED power state. The LED power will remain in the state until the default time-out period expires. *state* needs to be LED_ON or LED_OFF.

The function returns TRUE if state is valid and FALSE if it is not.

Notes

The LED POWER switch also controls the LED power. A user may override the setting made by this function.

The **ledSetDefault** function sets the default state of the LED power. This state overrides the value set by this function.

See Also

ledPowerSwitch, ledGetDefault, ledSetDefault

ledPowerSwitch

Read State of the LED Power Switch

Syntax

```
#include <ctools.h>
unsigned ledPowerSwitch(void);
```

Description

The ledPowerSwitch function returns the status of the led power switch. The function returns FALSE if the switch is released and TRUE if the switch is pressed.

Notes

This switch may be used by the program for user input. However, pressing the switch will have the side effect of changing the LED power state.

See Also

ledPower, ledSetDefault, ledGetDefault

ledSetDefault

Set Default Parameters for LED Power Control

Syntax

```
#include <ctools.h>
unsigned ledSetDefault(struct ledControl_tag ledControl);
```

Description

The **ledSetDefault** routine sets default parameters for LED power control. The controller controls LED power to 5000 I/O modules. To conserve power, the LEDs can be disabled.

The LED power setting can be changed by the user with the LED POWER switch on the controller. The LED power returns to its default state after a user specified time period.

The *ledControl* structure contains the default values. Refer to the **Structures and Types** section for a description of the fields in the *ledControl_tag* structure. Valid values for the *state* field are LED_ON and LED_OFF. Valid values for the *time* field are 1 to 65535 minutes.

The function returns TRUE if the parameters are valid and false if they are not. If either parameter is not valid, the default values are not changed.

The IO_SYSTEM resource needs to be requested before calling this function.

Example

```
#include <ctools.h>

void main(void)
{
       struct ledControl_tag ledControl;

       request_resource(IO_SYSTEM);

       /* Turn LEDS off after 20 minutes */
       ledControl.time  = 20;
       ledControl.state = LED_OFF;
       ledSetDefault(ledControl);

       release_resource(IO_SYSTEM);

       /* ... the reset of the program */
}
```

load

Read Parameters from EEPROM

Syntax

```
#include <ctools.h>
void load(unsigned section);
```

Description

The **load** function reads data from the specified *section* of the EEPROM into RAM.. Valid values for *section* are **EEPROM_EVERY** and **EEPROM_RUN**.

The **save** function writes data to the EEPROM.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

The **EEPROM_EVERY** section is not used.

The **EEPROM_RUN** section is loaded from EEPROM to RAM when the controller is reset and the Run/Service switch is in the RUN position. Otherwise default information is used for this section. This section contains:

- serial port configuration tables
- protocol configuration tables

See Also

save

master_message

Send Protocol Command

Syntax

```
#include <ctools.h>
extern unsigned master_message(FILE *stream, unsigned function,
unsigned slave_station, unsigned slave_address, unsigned
master_address, unsigned length);
```

Description

- The **master_message** function sends a command using a communication protocol. The communication protocol task waits for the response from the slave station. The current task continues execution.
- *stream* specifies the serial port.
- *function* specifies the protocol function code. Refer to the communication protocol manual for supported function codes.
- *slave* specifies the network address of the slave station. This is also known as the slave station number.
- *address* specifies the location of data in the slave station. Depending on the protocol function code, data may be read or written at this location.
- *master_address* specifies the location of data in the master (this controller). Depending on the protocol function code, data may be read or written at this location.
- *length* specifies the number or registers.

The **master_message** function returns the command status from the protocol driver.

Value	Description
MM_SENT	message transmitted to slave
MM_BAD_FUNCTION	function is not recognized
MM_BAD_SLAVE	slave station number is not valid
MM_BAD_ADDRESS	slave or master database address not valid
MM_BAD_LENGTH	too many or too few registers specified
MM_EOT	Master message status: DF1 slave response was an EOT message
MM_WRONG_RSP	Master message status: DF1slave response did not match command sent.
MM_CMD_ACKED	Master message status: DF1half duplex command has been acknowledged by slave – Master may now send poll command.

MM_EXCEPTION_FUNCTION	Master message status: Modbus slave returned a function exception.
MM_EXCEPTION_ADDRESS	Master message status: Modbus slave returned an address exception.
MM_EXCEPTION_VALUE	Master message status: Modbus slave returned a value exception.
MM_RECEIVED	Master message status: response received.
MM_RECEIVED_BAD_LENGTH	Master message status: response received with incorrect amount of data.

The calling task monitors the status of the command sent using the **get_protocol_status** function. The command field of the prot_status structure is set to **MM_SENT** if a master message is sent. It will be set to **MM_RECEIVED** when the response to the message is received with the proper length. It will be set to **MM_RECEIVED_BAD_LENGTH** when a response to the message is received with the improper length.

Notes

Refer to the communication protocol manual for more information.

To optimize performance, minimize the length of messages on com3 and com4. Examples of recommended uses for com3 and com4 are for local operator display terminals, and for programming and diagnostics using the Telepace program.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

Error! Reference source not found.

Example Using Modbus Protocol

This program sends a master message, on **com2**, using the Modbus protocol, then waits for a response from the slave. The number of good and failed messages is printed to **com1**.

```
/* --------------------------------------------
   poll.c
   Polling program for Modbus slave.
   -------------------------------------------- */

#include <ctools.h>

/* --------------------------------------------
   wait_for_response

   The wait_for_response function waits for a
   response to be received to a master_message on
   the serial port specified by stream. It returns
   when a response is received, or when the period
```

```
   specified by time (in tenths of a second)
   expires.
   ------------------------------------------- */

void wait_for_response(FILE *stream, unsigned time)
{
      struct prot_status status;
      static unsigned long good, bad;

      interval(0, 1);
      settimer(0, time);
      do {
            /* Allow other tasks to execute */
            release_processor();

            status = get_protocol_status(stream);
      }
      while (timer(0) && status.command == MM_SENT);

      if (status.command == MM_RECEIVED)
            good++;
      else
            bad++;
      fprintf(com1, "Good: %8lu  Bad: %8lu\r", good,
                  bad);
}
/* -------------------------------------------
      main

   The main function sets up serial ports then
   sends commands to a Modbus slave.
   ------------------------------------------- */
void main(void)
{
      struct prot_settings settings;
      struct pconfig portset;

      request_resource(IO_SYSTEM);

      /* disable protocol on serial port 1 */
      settings.type = NO_PROTOCOL;
      settings.station = 1;
      settings.priority = 3;
      settings.SFMessaging = FALSE;
      set_protocol(com1, &settings);

      /* Set communication parameters for port 1 */
      portset.baud      = BAUD9600;
      portset.duplex    = FULL;
      portset.parity    = NONE;
      portset.data_bits = DATA8;
      portset.stop_bits = STOP1;
      portset.flow_rx   = DISABLE;
      portset.flow_tx   = DISABLE;
      portset.type      = RS232;
      portset.timeout   = 600;
```

```
        set_port(com1, &portset);

        /* enable Modbus protocol on serial port 2 */
        settings.type = MODBUS_ASCII;
        settings.station = 2;
        settings.priority = 3;
        settings.SFMessaging = FALSE;
        set_protocol(com2, &settings);

        /* Set communication parameters for port 2 */
        portset.baud      = BAUD9600;
        portset.duplex    = HALF;
        portset.parity    = NONE;
        portset.data_bits = DATA8;
        portset.stop_bits = STOP1;
        portset.flow_rx   = DISABLE;
        portset.flow_tx   = DISABLE;
        portset.type      = RS485_2WIRE;
        portset.timeout   = 600;
        set_port(com2, &portset);

        release_resource(IO_SYSTEM);

        /* Main communication loop */
        while (TRUE)
        {
                /* Transfer slave inputs to outputs */
                request_resource(IO_SYSTEM);
                master_message(com2, 2, 1, 10001, 17, 8);
                release_resource(IO_SYSTEM);
                wait_for_response(com2, 10);

                /* Transfer inputs to slave outputs */
                request_resource(IO_SYSTEM);
                master_message(com2, 15, 1, 1, 10009, 8);
                release_resource(IO_SYSTEM);
                wait_for_response(com2, 10);

                /* Allow other tasks to execute */
                release_processor();
        }
}
```

Examples using DF1 Protocol

Full Duplex

Using the same example program above, apply the following calling format for the master_message function.

This code fragment uses the protected write command (*function*=0) to transmit 13 (*length*=13) 16-bit registers to slave station 10 (*slave*=10). The data will be read from registers 127 to 139 (*master_address*=127), and stored into registers

180 to 192 (*address*=180) in the slave station. The command will be transmitted on com2 (*stream*=com2).

master_message(com2, 0, 10, 180, 127, 13);

This code fragment uses the unprotected read command (*function*=1) to read 74 (*length*=74) 16-bit registers from slave station 37 (*slave*=37). The data will be read from registers 300 to 373 in the slave (*address*=300), and stored in registers 400 to 473 in the master (*master_address*=400). The command will be transmitted on com2 (*stream*=com2).

master_message(com2, 1, 37, 300, 400, 74);

This code fragment will send specific bits from a single 16-bit register in the master to slave station 33. The unprotected bit write command (*function*=5) will be used. Bits 0,1,7,12 and 15 of register 100 (*master_address*=100) will be sent to register 1432 (*address*=1432) in the slave. The *length* parameter is used as a bit mask and is evaluated as follows:

it mask	= 1001 0000 1000 0011 in binary	
	= 9083	in hexadecimal
	= 36,995	in decimal

Therefore the command, sent on com2, is:

master_message(com2, 5, 33, 1432, 100, 36995);

Half Duplex

The example program is the same as for Full Duplex except that instead of waiting for a response after calling master_message, the slave needs to be polled for a response. Add the following function **poll_for_response** to the example program above and call it instead of wait_for_response:

```
/* --------------------------------------------
   poll_for_response

   The poll_for_response function polls the
   specified slave for a response to a master
   message sent on the serial port specified by
   stream. It returns when the correct response
   is received, or when the period specified by
   time (in tenths of a second) expires.
   -------------------------------------------- */
unsigned poll_for_response(FILE *stream, unsigned slave, unsigned
time)
{
       struct prot_status status;
       unsigned done;
       static unsigned long good, bad;

       /* set timeout timer */
       interval( 0, 10 );
       settimer( 0, time );
```

```
        do
        {
                /* wait until command status changes or
                timer expires */
                do
                {
                        status = get_protocol_status( stream );
                        release_processor();
                }
                while(timer(0)&& (status.command==MM_SENT));

                /* command has been ACKed, send poll */
                if (status.command == MM_CMD_ACKED)
                {
                        pollABSlave(stream, slave);
                        done = FALSE;
                }

                /* response/command mismatch, poll again */
                else if (status.command == MM_WRONG_RSP)
                {
                        pollABSlave(stream, slave);
                        done = FALSE;
                }

                /* correct response was received */
                else if (status.command == MM_RECEIVED)
                {
                        good++;
                        done = TRUE;
                }

                /* timer has expired or status is MM_EOT */
                else
                {
                        bad++;
                        done = TRUE;
                }
        } while (!done);

        fprintf(com1, "Good: %8lu  Bad: %8lu\r", good,
        bad);
}
```

modbusExceptionStatus

Set Response to Protocol Command

Syntax

```
#include <ctools.h>
void modbusExceptionStatus(unsigned char status);
```

Description

The **modbusExceptionStatus** function is used in conjunction with the Modbus compatible communication protocol. It sets the result returned in response to the Read Exception Status command. This command is provided for compatibility with some Modbus protocol drivers for host computers.

The value of *status* is determined by the requirements of the host computer.

Notes

The specified result will be sent each time that the protocol command is received, until a new result is specified.

The result is cleared when the controller is reset. The application program needs to initialize the status each time it is run.

See Also

modbusSlaveID

modbusSlaveID

Set Response to Protocol Command

Syntax

```
#include <ctools.h>
void modbusSlaveID(unsigned char *string, unsigned length);
```

Description

The **modbusSlaveID** function is used in conjunction with the Modbus compatible communication protocol. It sets the result returned in response to the Report Slave ID command. This command is provided for compatibility with some Modbus protocol drivers for host computers.

string points to a string of at least *length* characters. The contents of the string is determined by the requirements of the host computer. The string is not NULL terminated and may contain multiple NULL characters.

The *length* specifies how many characters are returned by the protocol command. *length* needs to be in the range 1 to **REPORT_SLAVE_ID_SIZE**. If *length* is too large only the first **REPORT_SLAVE_ID_SIZE** characters of the string will be sent in response to the command.

Notes

The specified result will be sent each time that the protocol command is received, until a new result is specified.

The function copies the data pointed to by *string*. *string* may be modified after the function is called.

The result is cleared when the controller is reset. The application program needs to initialize the salve ID string each time it is run.

See Also

modbusExceptionStatus

modbusProcessCommand Function

Process a Modbus command and return the response.

Syntax

```
#include <ctools.h>
BOOLEAN processModbusCommand(
       FILE * stream,
       UCHAR * pCommand,
       UINT16 commandLength,
       UINT16 responseSize,
       UCHAR * pResponse,
       UINT16 * pResponseLength
       )
```

Description

The processModbusCommand function processes a Modbus protocol command and returns the response. The function can be used by an application to encapsulate Modbus RTU commands in another protocol.

stream is a FILE pointer that identifies the serial port where the command was received. This is used for to accumulate statistics for the serial port.

pCommand is a pointer to a buffer containing the Modbus command. The contents of the buffer needs to be a standard Modbus RTU message. The Modbus RTU checksum is not required.

commandLength is the number of bytes in the Modbus command. The length needs to include all the address and data bytes and not include the checksum bytes, if any, in the command buffer.

responseSize is the size of the response buffer in bytes. A 300-byte buffer is recommended. If this is not practical in the application, a smaller buffer may be supplied. Some responses may be truncated if a smaller buffer is used.

pResponse is a pointer to a buffer to contain the Modbus response. The function will store the response in this buffer in standard Modbus RTU format including two checksum bytes at the end of the response.

pResponseLength is a pointer to a variable to hold response length. The function will store the number of bytes in the response in this variable. The length will include two checksum bytes.

The function returns TRUE if the response is valid and can be used. It returns FALSE if the response is too long to fit into the supplied response buffer.

Notes

To use the function on a serial port, a protocol handler needs to be created for the encapsulating protocol. Set the protocol type for the port to NO_PROTOCOL to allow the custom handler to be used.

The function supports standard and extended addressing. Configure the protocol settings for the serial port for the appropriate protocol.

The Modbus RTU checksum is not required in the command so the encapsulating protocol may omit them if they are not needed. This may be useful in host devices that don't create a Modbus RTU message with checksum prior to encapsulation.

The Modbus RTU checksum is included in the response to support encapsulating a complete Modbus RTU format message. If the checksum is not needed by the encapsulating protocol the checksum bytes may be ignored.

See Also

set_protocol

Example

This example is taken from a protocol driver than encapsulates Modbus RTU messages in another protocol. It shows how to pass the Modbus RTU command to the Modbus driver, and obtain the response.

The example assumes the Modbus RTU messages are transmitted with the checksum. The length of the checksum is subtracted when calling the processModbusCommand function. The checksum is included when responding.

```
/* receive the packet in the encapsulating protocol */
/* verify the packet is valid */

/* locate the Modbus RTU command in the command buffer */
pCommandData = commandBuffer + PROTOCOL_HEADER_SIZE;

/* get length of Modbus RTU command from the packet header */
commandLength = commandBuffer[DATA_SIZE] - 2;

/* locate the Modbus RTU response in the response buffer leaving
room for the packet header */
pResponseData = responseBuffer + PROTOCOL_HEADER_SIZE;

/* process the Modbus message */
if (processModbusCommand(
       stream,
       pCommandData,
       commandLength,
       MODBUS_BUFFER_SIZE,
       pResponseData,
       &responseLength))
{
       /* put the response length in the header */
       responseBuffer[DATA_SIZE] = responseLength;

       /* fill in rest of packet header */
       /* transmit the encapsulated response */
}
```

modemAbort

Unconditionally Terminate Dial-up Connection

Syntax

```
#include <ctools.h>
void modemAbort(FILE *port);
```

Description

The **modemAbort** function unconditionally terminates a dial-up connection, connection in progress or modem initialization started by the C application. *port* specifies the serial port the where the modem is installed.

The connection or initialization is terminated only if it was started from a C application. Connections made from a Ladder Logic application and answered calls are not terminated.

This function can be used in a task exit handler.

Notes

The serial port type needs to be set to RS232_MODEM.

Note that a pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

Use this function in a task exit handler to clean-up any open dial-up connections or modem initializations. If a task is ended by executing end_task from another task, modem connections or initializations needs to be aborted in the exit handler. Otherwise, the reservation ID for the port remains valid. No other task or Ladder Logic program may use modem functions on the port. Not calling **modemAbort** or **modemAbortAll** in the task exit handler may result in the port being unavailable to any programs until the controller is reset.

The modem connection or initialization is automatically terminated when Telepace stops the C application and when the controller is rebooted.

All reservation IDs returned by the **modemDial** and **modemInit** functions on this port are invalid after calling **modemAbort**.

See Also

modemAbortAll, modemDial, modemDialEnd, modemDialStatus, modemInit, modemInitEnd, modemInitStatus, modemNotification

Example

Refer to the examples in the **Functions Overview** section.

modemAbortAll

Unconditionally Terminate All Dial-up Connections

Syntax

```
#include <ctools.h>
void modemAbort(void);
```

Description

The **modemAbortAll** function unconditionally terminates all dial-up connections, connections in progress or modem initializations started by the C application.

The connections or initializations are terminated only if they were started from a C application. Connections made from a Ladder Logic application and answered calls are not terminated.

This function can be used in a task exit handler.

Notes

A pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

Use this function in a task exit handler to clean-up any open dial-up connections or modem initializations. If executing end_task from another task ends a task, modem connections or initializations need to be aborted in the exit handler. Otherwise, the reservation ID for the port remains valid. No other task or Ladder Logic program may use modem functions on the port. Failing to call **modemAbort** or **modemAbortAll** in the task exit handler may result in the port being unavailable to any programs until the controller is reset.

The modem connection or initialization is automatically terminated when Telepace stops the C application and when the controller is rebooted.

This function will terminate all open dial-up connections or modem initializations started by the C application - even those started by other tasks. The exit handler can call this function instead of multiple calls to **modemAbort** if all the connections or initializations were started from the same task.

Reservation IDs returned by the **modemDial** and **modemInit** functions are invalid after calling **modemAbort**.

See Also

modemDial, modemDialEnd, modemDialStatus, modemInit, modemInitEnd, modemInitStatus, modemNotification

Example

This program installs an exit handler for the main task that terminates any dial-up connections made by the task. This handler is not strictly necessary if Telepace ends the main task. However, it demonstrates how to use the modemAbortAll function and an exit handler for another task in a more complex program.

```
#include <ctools.h>

/* --------------------------------------------
   The shutdown function aborts any active
   modem connections when the task is ended.
   -------------------------------------------- */
void shutdown(void)
{
       modemAbortAll();
}

void main(void)
{
       TASKINFO taskStatus;

       /* set up exit handler for this task */
       taskStatus = getTaskInfo(0);
       installExitHandler(taskStatus.taskID, shutdown);

       while(TRUE)
       {
              /* rest of main task here */

              /* Allow other tasks to execute */
              release_processor();
       }
}
```

modemDial

Connect to a Remote Dial-up Controller

Syntax

```
#include <ctools.h>
enum DialError modemDial(struct ModemSetup *configuration,
reserve_id *id);
```

Description

The **modemDial** function connects a controller to a remote controller using an external dial-up modem. One **modemDial** function may be active on each serial port. The **modemDial** function handles all port sharing and multiple dialing attempts.

The *ModemSetup* structure specified by *configuration* defines the serial port, dialing parameters, modem initialization string and the phone number to dial. Refer to the **Structures and Types** section for a description of the fields in the *ModemSetup* structure.

id points to a reservation identifier for the serial port. The identifier ensures that no other modem control function can access the serial port. This parameter needs to be supplied to the **modemDialEnd** and **modemDialStatus** functions.

The function returns an error code. DE_NoError indicates that the connect operation has begun. Any other code indicates an error. Refer to the *dialup.h* section for a complete description of error codes.

Notes

The serial port type must be set to RS232_MODEM.

The SCADAPack 100 does not support dial up connections on com port 1.

The **modemDialStatus** function returns the status of the connection attempt initiated by modemDial.

The **modemDialEnd** function terminates the connection to the remote controller. A pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

If a communication protocol is active on the serial port when a connection is initiated, the protocol will be disabled until the connection is made, then re-enabled. This allows the controller to communicate with the external modem on the port. The protocol settings will also be restored when a connection is terminated with the **modemDialEnd** function.

If a **modemInit** function or an incoming call is active on the port, the **modemDial** function cannot access the port and will return an error code of DE_NotInControl. If communication stops for more than five minutes, then outgoing call requests are allowed to end the incoming call. This prevents problems with the modem or the calling application from permanently disabling outgoing calls.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port.

To optimize performance, minimize the length of messages on com3 and com4. Examples of recommended uses for com3 and com4 are for local operator display terminals, and for programming and diagnostics using the Telepace program.

See Also

modemAbortAll, modemDialEnd, modemDialStatus, modemInit, modemInitEnd, modemInitStatus, modemNotification

Example

Refer to the examples in the **Functions Overview** section.

modemDialEnd

Terminate Dial-up Connection

Syntax

```
#include <ctools.h>
void modemDialEnd(FILE *port, reserve_id id, enum DialError
*error);
```

Description

The **modemDialEnd** function terminates a dial-up connection or connection in progress. *port* specifies the serial port the where the modem is installed. *id* is the port reservation identifier returned by the modemDial function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

Notes

The serial port type needs to be set to RS232_MODEM.

A connection can be terminated by any of the following events. Once terminated another modem function or incoming call can take control of the serial port.

- Execution of the modemDialEnd function.
- Execution of the modemAbort or modemAbortAll functions.
- The remote device hangs up the phone line.
- An accidental loss of carrier occurs due to phone line problems.

A pause of a few seconds is required between terminating a connection and initiating a new call. This pause allows the external modem time to hang up.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port. The **modemDialEnd** function returns a DE_NotInControl error code, if another modem function or incoming call is in control of the port.

This function cannot be called in a task exit handler. Use modemAbort instead.

See Also

modemAbortAll, modemDial, modemDialStatus, modemInit, modemInitEnd, modemInitStatus, modemNotification

modemDialStatus

Return Status of Dial-up Connection

Syntax

```
#include <ctools.h>
void modemDialStatus(FILE *port, reserve_id id, enum DialError *
error, enum DialState *state);
```

Description

The **modemDialStatus** function returns the status of a remote connection initiated by the **modemDial** function. *port* specifies the serial port where the modem is installed. *id* is the port reservation identifier returned by the **modemDial** function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

The function sets the variable pointed to by *state* to the current execution state of dialing operation. The state value is not valid if the error code is DE_NotInControl. Refer to the *dialup.h* section for a complete description of state codes.

Notes

The serial port type needs to be set to RS232_MODEM.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port. The **modemDialStatus** function will return a DE_NotInControl error code, if another dial function or incoming call is now in control of the port.

This function cannot be called in a task exit handler.

modemInit

Initialize Dial-up Modem

Syntax

```
#include <ctools.h>
enum DialError modemInit(struct ModemInit *configuration,
reserve_id *id);
```

Description

The **modemInit** function sends an initialization string to an external dial-up modem. It is typically used to set up a modem to answer incoming calls. One modemInit function may be active on each serial port. The modemInit function handles all port sharing and multiple dialing attempts.

The ModemInit structure pointed to by *configuration* defines the serial port and modem initialization string. Refer to the **Structures and Types** section for a description of the fields in the *ModemInit* structure.

The *id* variable is set to a reservation identifier for the serial port. The identifier ensures that no other modem control function can access the serial port. This parameter needs to be supplied to the modemInitEnd and modemInitStatus functions.

The function returns an error code. DE_NoError indicates that the initialize operation has begun. Any other code indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

Notes

The serial port type needs to be set to RS232_MODEM.

The **modemInitStatus** function returns the status of the connection attempt initiated by modemInit.

The **modemInitEnd** function terminates initialization of the modem.

If a communication protocol is active on the serial port, the protocol will be disabled until the initialization is complete then re-enabled. This allows the controller to communicate with the external modem on the port. The protocol settings will also be restored when initialization is terminated with the **modemInitEnd** function.

If a **modemDial** function or an incoming call is active on the port, the **modemInit** function cannot access the port and will return an error code of DE_NotInControl.

The reservation identifier is valid until the call is terminated and another modem function or an incoming call takes control of the port.

To optimize performance, minimize the length of messages on com3 and com4. Examples of recommended uses for com3 and com4 are for local operator display terminals, and for programming and diagnostics using the Telepace program.

This function cannot be called in a task exit handler.

See Also

modemAbortAll, modemDial, modemDialEnd, modemDialStatus, modemInitEnd, modemInitStatus, modemNotification

Example

Refer to the example in the **Functions Overview** section.

modemInitEnd

Abort Initialization of Dial-up Modem

Syntax

```
#include <ctools.h>
void modemInitEnd(FILE *port, reserve_id id, enum DialError
*error);
```

Description

The **modemInitEnd** function terminates a modem initialization in progress. *port* specifies the serial port where the modem is installed. *id* is the port reservation identifier returned by the **modemInit** function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the *dialup.h* section for a complete description of error codes.

Notes

The serial port type must be set to RS232_MODEM.

Normally this function should be called once the **modemInitStatus** function indicates the initialization is complete.

The reservation identifier is valid until the initialization is complete or terminated, and another modem function or an incoming call takes control of the port. The **modemInitEnd** function returns a DE_NotInControl error code, if another modem function or incoming call is in control of the port.

This function cannot be called in a task exit handler. Use modemAbort instead.

See Also

modemAbortAll, modemDial, modemDialEnd, modemDialStatus, modemInit, modemInitStatus, modemNotification

modemInitStatus

Return Status of Dial-up Modem Initialization

Syntax

```
#include <ctools.h>
void modemInitStatus(FILE *port, reserve_id id, enum DialError
*error, enum DialState *state);
```

Description

The **modemInitStatus** function returns the status a modem initialization started by the **modemInit** function. *port* specifies the serial port where the modem is installed. *id* is the port reservation identifier returned by the **modemInit** function.

The function sets the variable pointed to by *error*. If no error occurred DE_NoError is returned. Any other value indicates an error. Refer to the **Structures and Types** section for a complete description of error codes.

The function sets the variable pointed to by *state* to the current execution state of dialing operation. The state value is not valid if the error code is DE_NotInControl. Refer to the *dialup.h* section for a complete description of state codes.

Notes

The serial port type needs to be set to RS232_MODEM.

The port will remain in the DS_Calling state until modem initialization is complete or fails. The application should wait until the state is not DS_Calling before calling the **modemInitEnd** function.

The reservation identifier is valid until the initialization is complete or terminated, and another modem function or an incoming call takes control of the port.

This function cannot be called in a task exit handler.

See Also

modemAbortAll, modemDial, modemDialEnd, modemDialStatus, modemInit, modemInitEnd, modemNotification

modemNotification

Notify the modem handler of an important event

Syntax

```
#include <ctools.h>
void modemNotification(UINT16 port_index);
```

Description

The modemNotification function notifies the dial-up modem handler that an interesting event has occurred. This informs the modem handler not to disconnect an incoming call when an outgoing call is requested with modemDial.

This function is used with custom communication protocols. The function is usually called when a message is received by the protocol, although it can be called for other reasons.

The port_index indicates the serial port that received the message.

Notes

The serial port type needs to be set to RS232_MODEM.

Use the portIndex function to obtain the index of the serial port.

The dial-up connection handler prevents outgoing calls from using the serial port when an incoming call is in progress and communication is active. If communication stops for more than five minutes, then outgoing call requests are allowed to end the incoming call. This prevents the modem or the calling application from permanently disabling outgoing calls.

The function is used with programs that dial out through an external modem using the modemDial function. It is not required where the modem is used for dialing into the controller only.

See Also

modemAbortAll, modemDial, modemDialEnd, modemDialStatus, modemInit, modemInitEnd, modemInitStatus

off

Test Digital I/O Bit

Syntax

```
#include <ctools.h>
int off(unsigned channel, unsigned bit);
```

Description

The **off** function tests the status of the digital I/O point at *channel* and *bit*. *channel* must be in the range 0 to **DIO_MAX**. *bit* must be in the range 0 to 7.

The **off** function returns **TRUE** if the bit is off, **FALSE** if the bit is on, and –1 if *channel* or *bit* is invalid.

Notes

The **off** function may be used to check the status of digital inputs, outputs and configuration tables.

Use offsets from the symbolic constants DIN_START, DIN_END, DOUT_START and DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioRead8Din** directly.

See Also

ioRead8Din, turnoff, turnon, on

Example

This code fragment inverts the digital output point at the first digital output channel, bit 3.

```
request_resource(IO_SYSTEM);
if (off(DOUT_START, 3))
      turnon(DOUT_START, 3);
else
      turnoff(DOUT_START, 3);
release_resource(IO_SYSTEM);
```

on

Test Digital I/O Bit

Syntax

```
#include <ctools.h>
int on(unsigned channel, unsigned bit);
```

Description

The **on** function tests the status of the digital I/O point at *channel* and *bit*. *channel* needs to be in the range 0 to **DIO_MAX**. *bit* needs to be in the range 0 to 7.

The **on** function returns **TRUE** if the bit is on, **FALSE** if the bit is off, and –1 if *channel* or *bit* is invalid.

Notes

The **on** function may be used to check the status of digital inputs, outputs and configuration tables.

Use offsets from the symbolic constants DIN_START, DIN_END, DOUT_START and DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioRead8Din** directly.

See Also

ioRead8Din, turnoff, turnon, off

optionSwitch

Read State of Controller Option Switches

Syntax

```
#include <ctools.h>
unsigned optionSwitch(unsigned option);
```

Description

The **optionSwitch** function returns the state of the controller option switch specified by *option*. *option* may be 1, 2 or 3.

The function returns OPEN if the switch is in the open position. It returns CLOSED if the switch is in the closed position.

Notes

The option switches are located under the cover of the controller module. The SCADAPack LP, SCADAPack 100 and SCADAPack 4000 of controllers do not have option switches.

All options are user defined.

However, when a SCADAPack I/O module is placed in the Register Assignment, option switch 1 selects the input range for analog inputs on this module. When the SCADAPack AOUT module is placed in the Register Assignment, option switch 2 selects the output range for analog outputs on this module. Refer to the ***SCADAPack System Hardware Manual*** for further information on option switches.

overrideDbase

Overwrite Value in Forced I/O Database

Syntax

```
#include <ctools.h>
unsigned overrideDbase(unsigned type, unsigned address, int
value);
```

Description

The **overrideDbase** function writes *value* to the I/O database even if the database register is currently forced. *type* specifies the method of addressing the database. *address* specifies the location in the database.

If the register is currently forced, the register remains forced but forced to the new *value.*

If the *address* or addressing *type* is not valid, the I/O database is left unchanged and FALSE is returned; otherwise TRUE is returned. The table below shows the valid address types and ranges.

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL	1 bit
	10001 to 10000 + NUMSTATUS	1 bit
	30001 to 30000 + NUMINPUT	16 bit
	40001 to 40000 + NUMHOLDING	16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

When writing to LINEAR digital addresses, *value* is a bit mask which writes data to 16 1-bit registers at once.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

Refer to the **Functions Overview** chapter for more information.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

setdbase, setForceFlag

Example

```
#include <ctools.h>
void main(void)
{
       request_resource(IO_SYSTEM);
```

```
        overrideDbase(MODBUS, 40001, 102);
        overrideDbase(LINEAR, 302, 330);

        release_resource(IO_SYSTEM);
}
```

pidExecute

Execute PID control algorithm

Syntax

```
#include <ctools.h>
BOOLEAN pidExecute(PID_DATA * pData);
```

Description

This function executes the PID algorithm. The function may be called as often as desired, but needs to be called at least once per the value in the period field for proper operation.

The function has one parameter. *pData* is a pointer to a structure containing the PID block data and outputs.

The function returns TRUE if the PID block executed. The function returns FALSE if it was not time for execution.

Notes

To properly initialize the PID algorithm do one of the following.

Call the **pidInitialize** function once before calling this function the first time, or

put the PID algorithm in manual mode (autoMode = FALSE in PID_DATA) for the first call to the **pidExecute** function.

See Also

pidInitialize

Example

This example initializes one PID control structure and executes the control algorithm continuously. Input data is read from analog inputs. Output data is written to analog outputs.

```
#include <ctools.h>

// event number to signal when I/O scan completes
#define IO_COMPLETE 0

void main(void)
{
	INT16 ainData[4];		// analog input data
	INT16 aoutData[4];	// analog output data
	PID_DATA pidData;		// PID algorithm data
	BOOLEAN executed;		// indicates if PID executed

	// read analog input
	ioRequest(MT_Ain4, 0);
	ioNotification(IO_COMPLETE);
	wait_event(IO_COMPLETE);
```

```
        ioReadAin4(0, ainData);

        // get initial process value from analog input
        pidData.pv = ainData[0];

        // configure PID block
        pidData.sp           = 1000;
        pidData.gain         = 1;
        pidData.reset        = 100;
        pidData.rate         = 0;
        pidData.deadband     = 10;
        pidData.fullScale    = 32767;
        pidData.zeroScale    = 0;
        pidData.manualOutput = 0;
        pidData.period       = 1000;
        pidData.autoMode     = TRUE;

        // initialize the PID block
        pidInitialize(&pidData);

        // main loop
        while (TRUE)
        {
                // execute all I/O requests
                ioRequest(MT_Ain4, 0);
                ioNotification(IO_COMPLETE);
                wait_event(IO_COMPLETE);

                // get process input
                ioReadAin4(0, ainData);
                pidData.pv = ainData[0];

                // execute the PID block
                executed = pidExecute(&pidData);

                // if the output changed
                if (executed)
                {
                        // write the output to analog output module
                        aoutData[0] = pidData.output;
                        ioWriteAout4(0, aoutData);
                        ioRequest(MT_Aout4, 0);
                }

                // release processor to other priority 1 tasks
                release_processor();
        }
}
```

pidInitialize

Initialize PID controller data

Syntax

```
#include <ctools.h>
void pidInitialize(PID_DATA * pData);
```

Description

This function initializes the PID algorithm data.

The function has one parameter. *pData* is a pointer to a structure containing the PID data and outputs.

The function should be called once before calling the **pidExecute** function for the first time. The structure pointed to by *pData* needs to contain valid values for sp, pv, and manualOutput before calling the function.

The function has no return value.

See Also

pidExecute

Example

See the example for **pidExecute**.

pollABSlave

Poll DF1 Slave for Response

Syntax

```
#include <ctools.h>
unsigned pollABSlave(FILE *stream, unsigned slave);
```

Description

The **pollABSlave** function is used to send a poll command to the slave station specified by *slave* in the DF1 Half Duplex protocol configured for the specified port. *stream* specifies the serial port.

The function returns **FALSE** if the slave number is invalid, or if the protocol currently installed on the specified serial port is not an DF1 Half Duplex protocol. Otherwise it returns **TRUE** and the protocol command status is set to **MM_SENT**.

Notes

See the example using the **pollABSlave** function in the sample polling function "poll_for_response" shown in the example for the **master_message** function.

See Also

master_message

Example

This program segment polls slave station 9 for a response communicating on the **com2** serial port.

```
#include <ctools.h>

pollABSlave(com2, 9);
```

poll_event

Test for Event Occurrence

Syntax

```
#include <ctools.h>
int poll_event(int event);
```

Description

The **poll_event** function tests if an event has occurred.

The **poll_event** function returns **TRUE,** and the event counter is decrements, if the event has occurred. Otherwise it returns **FALSE**.

The current task continues to execute.

Notes

Refer to the **Real Time Operating System** section for more information on events.

Valid events are numbered 0 to RTOS_EVENTS **-** 1. Any events defined in primitiv.h are not valid events for use in an application program.

See Also

signal_event, wait_event, startTimedEvent

Example

This program implements a somewhat inefficient transfer of data between **com1** and **com2**. (It would be more efficient to test for EOF from getc).

```
#include <ctools.h>

void main(void)
{
	while(TRUE)
	{
		if (poll_event(COM1_RCVR))
			fputc(getc(com1), com2);
		if (poll_event(COM2_RCVR))
			fputc(getc(com2), com1);

		/* Allow other tasks to execute */
		release_processor();
	}
}
```

poll_message

Test for Received Message

Syntax

```
#include <ctools.h>
envelope *poll_message(void);
```

Description

The **poll_message** function tests if a message has been received by the current task.

The **poll_message** function returns a pointer to an envelope if a message has been received. It returns **NULL** if no message has been received.

The current task continues to execute.

Notes

Refer to the **Real Time Operating System** section for more information on messages.

See Also

send_message, receive_message

Example

This task performs a function continuously, and processes received messages (from higher priority tasks) when they are received.

```
#include <ctools.h>

void task(void)
{
	envelope *letter;

	while(TRUE)
	{
		letter=poll_message();
		if (letter != NULL)
			/* process the message now */

		/* more code here */
	}
}
```

poll_resource

Test Resource Availability

Syntax

```
#include <ctools.h>
int poll_resource(int resource);
```

Description

The **poll_resource** function tests if the resource specified by *resource* is available. If the resource is available it is given to the task.

The **poll_resource** function returns **TRUE** if the resource is available. It returns **FALSE** if it is not available.

The current task always continues to execute.

Notes

Refer to the **Real Time Operating System** section for more information on resources.

See Also

request_resource, release_resource

portConfiguration

Get Pointer to Port Configuration Structure

Syntax

```
#include <ctools.h>
struct pconfig *portConfiguration(FILE *stream);
```

Description

The **portConfiguration** function returns a pointer to the configuration structure for *stream*. A NULL pointer is returned if *stream* is not valid.

Notes

It is recommended the **get_port** and **set_port** functions be used to access the configuration table.

portIndex

Get Index of Serial Port

Syntax

```
#include <ctools.h>
unsigned portIndex(FILE *stream);
```

Description

The **portIndex** function returns an array index for the serial port specified by *stream*. It will return a value suitable for an array index, in increasing order of external serial port numbers, if no error occurs.

If the stream is not recognized, SERIAL_PORTS is returned, to indicate an error.

See Also

portStream

portStream

Get Serial Port Corresponding to Index

Syntax

```
#include <ctools.h>
FILE *portStream(unsigned index);
```

Description

The **portStream** function returns the file pointer corresponding to *index*. This function is the inverse of the **portIndex** function. If the index is not valid, the NULL pointer is returned.

See Also

portIndex

processModbusCommand

Process a Modbus Command and Return the Response

Syntax

```
#include <ctools.h>
BOOLEAN processModbusCommand(
       FILE * stream,
       UCHAR * pCommand,
       UINT16 commandLength,
       UINT16 responseSize,
       UCHAR * pResponse,
       UINT16 * pResponseLength
       )
```

Description

The processModbusCommand function processes a Modbus protocol command and returns the response. The function can be used by an application to encapsulate Modbus RTU commands in another protocol.

stream is a FILE pointer that identifies the serial port where the command was received. This is used for to accumulate statistics for the serial port.

pCommand is a pointer to a buffer containing the Modbus command. The contents of the buffer needs to be a standard Modbus RTU message. The Modbus RTU checksum is not required.

commandLength is the number of bytes in the Modbus command. The length needs to include all the address and data bytes and not include the checksum bytes, if any, in the command buffer.

responseSize is the size of the response buffer in bytes. A 300-byte buffer is recommended. If this is not practical in the application, a smaller buffer may be supplied. Some responses may be truncated if a smaller buffer is used.

pResponse is a pointer to a buffer to contain the Modbus response. The function will store the response in this buffer in standard Modbus RTU format including two checksum bytes at the end of the response.

pResponseLength is a pointer to a variable to hold response length. The function will store the number of bytes in the response in this variable. The length will include two checksum bytes.

The function returns TRUE if the response is valid and can be used. It returns FALSE if the response is too long to fit into the supplied response buffer.

Notes

To use the function on a serial port, a protocol handler needs to be created for the encapsulating protocol. Set the protocol type for the port to NO_PROTOCOL to allow the custom handler to be used.

The function supports standard and extended addressing. Configure the protocol settings for the serial port for the appropriate protocol.

The Modbus RTU checksum is not required in the command so the encapsulating protocol may omit them if they are not needed. This may be useful in host devices that don't create a Modbus RTU message with checksum prior to encapsulation.

The Modbus RTU checksum is included in the response to support encapsulating a complete Modbus RTU format message. If the checksum is not needed by the encapsulating protocol the checksum bytes may be ignored.

See Also

setProtocolSettings

Example

This example is taken from a protocol driver than encapsulates Modbus RTU messages in another protocol. It shows how to pass the Modbus RTU command to the Modbus driver, and obtain the response.

The example assumes the Modbus RTU messages are transmitted with the checksum. The length of the checksum is subtracted when calling the processModbusCommand function. The checksum is included when responding.

```
/* receive the packet in the encapsulating protocol */
/* verify the packet is valid */

/* locate the Modbus RTU command in the command buffer */
pCommandData = commandBuffer + PROTOCOL_HEADER_SIZE;

/* get length of Modbus RTU command from the packet header */
commandLength = commandBuffer[DATA_SIZE] - 2;

/* locate the Modbus RTU response in the response buffer leaving
room for the packet header */
pResponseData = responseBuffer + PROTOCOL_HEADER_SIZE;

/* process the Modbus message */
if (processModbusCommand(
       stream,
       pCommandData,
       commandLength,
       MODBUS_BUFFER_SIZE,
       pResponseData,
       &responseLength))
{
       /* put the response length in the header */
       responseBuffer[DATA_SIZE] = responseLength;

       /* fill in rest of packet header */
       /* transmit the encapsulated response */
}
```

pulse

Generate a Square Wave

Syntax

```
#include <ctools.h>
void pulse(unsigned channel, unsigned bit, unsigned timer,
unsigned on, unsigned period);
```

Description

The **pulse** function generates a square wave with a specified duty cycle on a digital output point.

- *channel* specifies the digital output channel;
- *bit* specified the digital output bit;
- *timer* specifies the timer used to generate the square wave;
- *on* specifies the time the output will be on, measured in timer ticks;
- *period* specifies the period of the wave (on time plus off time), measured in timer ticks.

If an error occurs, the current task's error code is set as follows.

TIMER_BADTIMER	if the timer number is invalid
TIMER_BADVALUE	if the period is less than the on time
TIMER_BADADDR	if the digital channel or bit is invalid

Notes

The length of a timer tick is set with the **interval** function. The default value is 0.1 seconds.

To stop the square wave, set the *timer* to 0 with the **settimer** function. The square wave will stop if the controller is reset.

For an orderly start of the duty cycle, use the following sequence:

```
settimer(t, 0);            /* stop the timer */
request_resource(IO_SYSTEM);
turnoff(c, b);             /* start with a rising edge */
release_resource(IO_SYSTEM);
pulse(c, b, t, o, p);      /* start pulses */
```

If the specified I/O point is on when the **pulse** function is executed, the square wave will start with the off portion of the cycle.

Use the **timeout** function to generate irregular or non-repeating sequences.

Use offsets from the symbolic constants DIN_START, DIN_END, DOUT_START and DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioWrite8Dout** directly.

See Also

pulse_train, settimer, timeout, ioWrite8Dout

Example

This code fragment generates a 60% duty cycle output with a period of 5 seconds. Bit 7 of channel 3 is controlled. Timer 10 generates the square wave.

```
settimer(10, 0);           /* stop timer */

request_resource(IO_SYSTEM);
turnoff(3, 7);             /* turn off the bit  */
release_resource(IO_SYSTEM);

interval(10, 10);   /* set tick rate to 1.0 s */
pulse(3, 7, 10, 3, 5);           /* on = 60% of 5 = 3 */
```

pulse_train

Generate Finite Number of Pulses

Syntax

```
#include <ctools.h>
void pulse_train(unsigned channel, unsigned bit, unsigned timer,
unsigned pulses);
```

Description

The **pulse_train** function generates a specified number of pulses on a digital output point. The output is a square wave with a 50% duty cycle and a period of 200 milliseconds (5 Hz).

- *channel* specifies the digital output channel.
- *bit* specified the digital output bit.
- *timer* specifies the timer used to generate the square wave.
- *pulses* specifies the number of pulses. The timer interval acts as a multiplier of the number of pulses. The total number of pulses is *pulses* * interval.

If an error occurs, the current task's error code is set as follows.

TIMER_BADTIMER	if the timer number is invalid
TIMER_BADVALUE	if the period is less than the on time
TIMER_BADADDR	if the digital channel or bit is invalid

Notes

To stop the square wave, set the *timer* to 0 with the **settimer** function. The square wave will stop if the controller is reset.

For an orderly start to the pulses, use the following sequence:

```
settimer(t, 0);             /* stop the timer */
request_resource(IO_SYSTEM);
turnoff(c, b);              /* start with a rising edge */
release_resource(IO_SYSTEM);
pulse_train(c, b, timer, pulses);
```

Use offsets from the symbolic constants DIN_START, DIN_END, DOUT_START and DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioWrite8Dout** directly.

See Also

pulse, settimer, timeout, ioWrite8Dout

Example

This code fragment generates 300 pulses on channel 3, bit 4.

```
interval(2, 1);                          /* multiplier = 1 */
pulse_train(3, 4, 2, 300); /* 300 pulses */

This code fragment also generates 300 pulses on channel 3, bit 4.
It shows the use of a multiplier.
interval(2, 100);                /* multiplier = 100 */
pulse_train(3, 4, 2, 3);   /* 300 pulses */
```

queue_mode

Control Serial Data Transmission

Syntax

```
#include <ctools.h>
void queue_mode(FILE *stream, int mode);
```

Description

The **queue_mode** function controls transmission of the serial data. Normally data output to a serial port are placed in the transmit buffer and transmitted as soon as the hardware is ready. If queuing is enabled, the characters are held in the transmit buffer until queuing is disabled. If the buffer fills, queuing is disabled automatically.

stream specifies the serial port. If it is not valid the function has no effect.

mode specifies the queuing control. It may be **DISABLE** or **ENABLE**.

Notes

Queuing is often used with communication protocols that use character timing for message framing. Its uses in an application program are limited.

readCounter

Read Accumulator Input

Syntax

```
#include <ctools.h>
unsigned long readCounter(unsigned counter, unsigned clear);
```

Description

The readCounter routine reads the digital input counter specified by *counter*. The *counter* may be 0, 1 or 2. If *clear* is TRUE the counter is cleared after reading; otherwise if it is FALSE the counter continues to accumulate.

If counter is not valid, a BAD_COUNTER error is reported for the current task.

Notes

The three DIN/counter inputs are located on the SCADAPack, SCADAPack LP or SCADAPack 100. Refer to the ***System Hardware Manual*** for more information on the hardware.

The counter increments on the rising edge of the input signal.

See Also

readCounterInput, check_error

readCounterInput

Read Counter Input Status

Syntax

```
#include <ctools.h>
unsigned readCounterInput(unsigned input)
```

Description

The readCounterInput function returns the status of the DIN/counter input point specified by *input*. It returns TRUE if the input is ON and FALSE if the input is OFF.

If input is not valid, the function returns FALSE.

Notes

The three DIN/counter inputs are located on the 5203 or 5204 controller board. Refer to the ***System Hardware Manual*** for more information on the hardware.

See Also

readCounter

readBattery

Read Lithium Battery Voltage

Syntax

```
#include <ctools.h>
int readBattery(void);
```

Description

The **readBattery** function returns the RAM backup battery voltage in millivolts. The range is 0 to 5000 mV. A normal reading is about 3600 mV.

Example

```
#include <ctools.h>

if (readBattery() < 2500)
{
       fprintf(com1, "Battery Voltage is low\r\n");
       }
```

readInternalAD

Read Controller Internal Analog Inputs

Syntax

```
#include <ctools.h>
int readInternalAD(unsigned channel);
```

Description

The **readInternalAD** function reads analog inputs connected to the internal AD converter. *channel* may be 0 to 7.

The function returns a value in the range 0 to 32767.

Notes

There are only two channels with signals connected to them.

- AD_THERMISTOR reads the thermistor input.
- AD_BATTERY reads the battery input

See Also

readBattery

readStopwatch

Read Stopwatch Timer

Syntax

```
#include <ctools.h>
unsigned long readStopwatch(void)
```

Description

The **readStopwatch** function reads the stopwatch timer. The stopwatch time is in ms and has a resolution of 10 ms. The stopwatch time rolls over to 0 when it reaches the maximum value for an unsigned long integer: 4,294,967,295 ms (or about 49.7 days).

See Also

settimer, **timer**

Example

This program measures the execution time in ms of an operation.

```
#include <ctools.h>

void main(void)
{
	unsigned long startTime, endTime;

	startTime = readStopwatch();
	/* operation to be timed */
	endTime = readStopwatch();

	printf("Execution time = %lu ms\r\n", endTime - startTime);
}
```

readThermistor

Read Controller Ambient Temperature

Syntax

```
#include <ctools.h>
int readThermistor(unsigned scale);
```

Description

The **readThermistor** function returns the temperature measured at the main board in the specified temperature *scale*. If the temperature scale is not recognized, the temperature is returned in Celsius. The *scale* may be T_CELSIUS, T_FAHRENHEIT, T_KELVIN or T_RANKINE.

The temperature is rounded to the nearest degree.

Example

```
#include <ctools.h>

void checkTemperature(void)
{
	int temperature;

	temperature = readThermistor(T_FAHREHEIT);
	if (temperature < 0)
		fprintf(com1, “It’s COLD!!!\r\n”);
	else if (temperature > 90)
		fprintf(com1, “It’s HOT!!!\r\n”);
}
```

read_timer_info

Get Timer Status

Syntax

```
#include <ctools.h>
struct timer_info read_timer_info(unsigned timer);
```

Description

The **read_timer_info** function gets status information for the timer specified by *timer*.

The **read_timer_info** function returns a timer_info structure with information about the specified timer. Refer to the description of the timer_info structure for information about the fields.

See Also

settimer, pulse, pulse_train, timeout

Example

This program starts a pulse train and displays timer information.

```
#include <ctools.h>
void main(void)
{
       struct timer_info tinfo;

       /* Start Pulse Train */
       interval(10, 1);                         /* multiplier = 1
*/
       pulse_train(3, 5, 10, 500);
       while (timer(10) > 100)    /* wait a while */
       {
              /* Allow other tasks to execute */
              release_processor();
       }
       /* Display Status of Pulse Train */
       tinfo = read_timer_info(10);
       printf("Pulses Remaining: %d\r\n",
                     tinfo.time/2);
       printf("Output Channel:   %d\r\n",
                     tinfo.channel);
       printf("Output Bit:       %d\r\n", tinfo.bit);
}
```

receive_message

Receive a Message

Syntax

```
#include <ctools.h>
envelope *receive_message(void);
```

Description

The **receive_message** function reads the next available envelope from the message queue for the current task. If the queue is empty, the task is blocked until a message is sent to it.

The **receive_message** function returns a pointer to an envelope structure.

Notes

Refer to the **Real Time Operating System** section for more information on messages.

See Also

send_message, poll_message

Example

This task waits for messages, then prints their contents. The envelopes received are returned to the operating system.

```
#include <ctools.h>

void show_message(void)
{
	envelope *msg;
	while (TRUE)
	{
		msg = receive_message();
		printf("Message data %ld\r\n", msg->data);
		deallocate_envelope(msg);
	}
}
```

release_processor

Release Processor to other Tasks

Syntax

```
#include <ctools.h>
void release_processor(void);
```

Description

The **release_processor** function releases control of the CPU to other tasks. Other tasks of the same priority will run. Tasks of the same priority run in a round-robin fashion, as each releases the processor to the next.

Notes

The **release_processor** function needs to be called in all idle loops of a program to allow other tasks to execute.

Release all resources in use by a task before releasing the processor.

Refer to the **Real Time Operating System** section for more information on tasks and task scheduling.

See Also

release_resource

release_resource

Release Control of a Resource

Syntax

```
#include <ctools.h>
void release_resource(int resource);
```

Description

The **release_resource** function releases control of the resource specified by *resource*.

If other tasks are waiting for the resource, the highest priority of these tasks, is given the resource and is made ready to execute. If no tasks are waiting the resource is made available, and the current task continues to run.

Notes

Refer to the **Real Time Operating System** section for more information on resources.

See Also

request_resource, poll_resource

Example

See the example for the **request_resource** function.

report_error

Set Task Error Code

Syntax

```
#include <ctools.h>
void report_error(int error);
```

Description

The **report_error** functions sets the error code for the current task to *error*. An error code is maintained for each executing task.

Notes

This function is used in sharable I/O routines to return error codes to the task using the routine.

Some functions supplied with the Microtec C compiler report errors using the global variable **errno**. The error code in this variable may be written over by another task before it can be used.

See also

check_error

request_resource

Obtain Control of a Resource

Syntax

```
#include <ctools.h>
void request_resource(int resource);
```

Description

The **request_resource** function obtains control of the resource specified by *resource*. If the resource is in use, the task is blocked until it is available.

Notes

Use the **request_resource** function to control access to non-sharable resources. Refer to the **Real Time Operating System** section for more information on resources.

See Also

release_resource, poll_resource

Example

This code fragment obtains the dynamic memory resource, allocates some memory, and releases the resource.

```
#include <ctools.h>

void task(void)
{
	unsigned *ptr;

	/* ... code here */

	request_resource(DYNAMIC_MEMORY);
	ptr = (unsigned *)malloc((size_t)100);
	release_resource(DYNAMIC_MEMORY);

	/* ... more code here */
}
```

resetAllABSlaves

Erase All DF1 Slave Responses

Syntax

```
#include <ctools.h>
unsigned resetAllABSlaves(FILE *stream);
```

Description

The **resetAllABSlaves** function is used to send a protocol message to all slaves communicating on the specified port to erase all responses not yet polled. *stream* specifies the serial port.

This function applies to the DF1 Half Duplex protocols only. The function returns **FALSE** if the protocol currently installed on the specified serial port is not an DF1 Half Duplex protocol, otherwise it returns **TRUE**.

Notes

The purpose of this command is to re-synch slaves with the master if the master has lost track of the order of responses to poll. This situation may exist if the master has been power cycled, for example. This function should not normally be needed if polling is done using the sample polling function "poll_for_response" shown in the example for the **master_message** function.

Example

This program segment will cause all slaves communicating on the **com2** serial port to erase all pending responses.

```
#include <protocol.h>


resetAllABSlaves(com2);
```

resetClockAlarm

Acknowledge and Reset Real Time Clock Alarm

Syntax

```
#include <ctools.h>
void resetClockAlarm(void);
```

Description

Real time clock alarms occur once after being set. The alarm setting remains in the real time clock. The alarm needs to be acknowledged before it can occur again.

The **resetClockAlarm** function acknowledges the last real time clock alarm and re-enables the alarm. Calling the function after waking up from an alarm will reset the alarm for 24 hours after the current alarm.

Notes

This function should be called after a real time clock alarm occurs. This includes after returning from the **sleep** function with a return code of WS_REAL_TIME_CLOCK.

The alarm time is not changed by this function.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

setClockAlarm, getClockAlarm, alarmIn

Example

See the example for the **installClockHandler** function.

route

Redirect Standard I/O Streams

Syntax

```
#include <ctools.h>
void route(FILE *logical, FILE *hardware);
```

Description

The **route** function redirects the I/O streams associated with stdout, stdin, and stderr. These streams are routed to the com1 serial port. *logical* specifies the stream to redirect. *hardware* specifies the hardware device which will output the data. It may be one of com1, com2, com3 or com4.

Notes

This function has a global effect, so all tasks need to agree on the routing.

Output streams need to be redirected to a device that supports output. Input streams need to be redirected to a device that supports input.

Example

This program segment will redirect all input, output and errors to the **com2** serial port.

```
#include <ctools.h>

route(stderr, com2);        /* send errors to com2 */
route(stdout, com2);        /* send output to com2 */
route(stdin, com2);         /* get input from com2 */
```

runLed

Control Run LED State

Syntax

```
#include <ctools.h>
void runLed(unsigned state);
```

Description

The **runLed** function sets the run light LED to the specified state. *state* may be one of the following values.

LED_ON turn on run LED
LED_OFF turn off run LED

The run LED remains in the specified state until changed, or until the controller is reset.

Notes

The ladder logic interpreter controls the state of the RUN LED. If ladder logic is installed in the controller, a C program should not use this function.

Example

```
#include <ctools.h>

void main(void)
{
       runLed(LED_ON);             /* program is running */
       /* ... the rest of the code */
}
```

save

Write Parameters to EEPROM

Syntax

```
#include <ctools.h>
void save(unsigned section);
```

Description

The **save** function writes data from RAM to the specified section of the EEPROM. Valid values for *section* are **EEPROM_EVERY** and **EEPROM_RUN**.

Notes

The **EEPROM_EVERY** section is loaded whenever the controller is reset. It is not used.

The **EEPROM_RUN** section is loaded from EEPROM to RAM when the controller is reset and the Run/Service switch is in the RUN position. Otherwise default information is used for this section. This section contains:

- serial port configuration tables
- protocol configuration tables
- store and forward enable flags
- LED power settings
- make for wake-up sources
- execution period on power-up for PID controllers
- HART modem settings

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

load

Example

This code fragment saves all parameters.

```
request_resource(IO_SYSTEM);
save(EEPROM_RUN);
release_resource(IO_SYSTEM);
```

send_message

Send a Message to a Task

Syntax

```
#include <ctools.h>
void send_message(envelope *penv);
```

Description

The **send_message** function sends a message to a task. The envelope specified by *penv* contains the message destination, type and data.

The envelope is placed in the destination task's message queue. If the destination task is waiting for a message it is made ready to execute.

The current task is not blocked by the **send_message** function.

Notes

Envelopes are obtained from the operating system with the **allocate_envelope** function.

See Also

receive_message, poll_message, allocate_envelope

Example

This program creates a task to display a message and sends a message to it.

```
#include <ctools.h>

void showIt(void)
{
	envelope *msg;

	while (TRUE)
	{
		msg = receive_message();
		printf("Message data %ld\r\n", msg->data);
		deallocate_envelope(msg);
	}
}

void main(void)
{
	envelope *msg;			/* message pointer */
	unsigned tid;			/* task ID */

	tid = create_task(showIt, 2, APPLICATION, 1);
	msg = allocate_envelope();
	msg->destination = tid;
	msg->type        = MSG_DATA;
	msg->data        = 1002;
```

```
        send_message(msg);

        /* wait for ever so that main and other
        APPLICATION tasks won't end */
        while(TRUE)
        {
              /* Allow other tasks to execute */
              release_processor();
        }
}
```

setABConfiguration

Set DF1 Protocol Configuration

Syntax

```
#include <ctools.h>
int setABConfiguration(FILE *stream, struct    ABConfiguration
*ABConfig);
```

Description

The **setABConfiguration** function sets DF1 protocol configuration parameters. *stream* specifies the serial port. *ABConfig* references an DF1protocol configuration structure. Refer to the description of the ABConfiguration structure for an explanation of the fields.

The **setABConfiguration** function returns **TRUE** if the settings were changed. It returns **FALSE** if *stream* does not point to a valid serial port.

Example

This code fragment changes the maximum protected address to 7000. This is the maximum address accessible by protected DF1 commands received on com2.

```
#include <ctools.h>
struct ABConfiguration ABConfig;

getABConfiguration(com2, &ABConfig);
ABConfig.max_protected_address = 7000;
setABConfiguration(com2, &ABConfig);
```

setBootType

Set Controller Boot Up State

Syntax

```
#include <ctools.h>
void setBootType(unsigned type);
```

Description

The **setBootType** function defines the controller boot up type code. This function is used by the operating system start up routines. It should not be used in an application program.

Notes

The value set with this function can be read with the **getBootType** function.

setclock

Set Real Time Clock

Syntax

```
#include <ctools.h>
void setclock(struct clock *now);
```

Description

The **setclock** function sets the real time clock. *now* references a clock structure containing the time and date to be set.

Refer to the **Structures and Types** section for a description of the fields. The fields of the clock structure need to be set with valid values for the clock to operate properly.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

getclock

Example

This function switches the clock to daylight savings time.

```
#include <ctools.h>
#include <primitiv.h>

void daylight(void)
{
	struct clock now;

	request_resource(IO_SYSTEM);
	now = getclock();
	now.hour = now.hour + 1 % 24;
	setclock(&now);
	request_resource(IO_SYSTEM);
}
```

setClockAlarm

Set the Real Time Clock Alarm

Syntax

```
#include <ctools.h>
unsigned setClockAlarm(ALARM_SETTING alarm);
```

Description

The **setClockAlarm** function configures the real time clock to alarm at the specified alarm setting. The ALARM_SETTING structure *alarm* specifies the time of the alarm. Refer to the *rtc.h* section for a description of the fields in the structure.

The function returns TRUE if the alarm can be configured, and FALSE if there is an error in the alarm setting. No change is made to the alarm settings if there is an error.

Notes

An alarm will occur only once, but remains set until disabled. Use the **resetClockAlarm** function to acknowledge an alarm that has occurred and re-enable the alarm for the same time.

Set the alarm type to AT_NONE to disable an alarm. It is not necessary to specify the hour, minute and second when disabling the alarm.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

alarmIn, getclock

Example

```
#include <ctools.h>

/* --------------------------------------------
   wakeUpAtEight

       The wakeUpAtEight function sets an alarm
       for 08:00 AM and puts the controller into
       sleep mode.
   -------------------------------------------- */

void wakeUpAtEight(void)
{
       ALARM_SETTING alarm;
       unsigned wakeSource;

       /* Set alarm for 08:00 */
       alarm.type   = AT_ABSOLUTE;
       alarm.hour   = 8;
```

```
        alarm.minute = 0;
        alarm.second = 0;

        /* Set the alarm */
        request_resource(IO_SYSTEM);
        setClockAlarm(alarm)
        release_resource(IO_SYSTEM);

        /* Sleep until alarm ignoring other wake ups */
        do
        {
                request_resource(IO_SYSTEM);
                wakeSource = sleep();
                release_resource(IO_SYSTEM);
        } until (wakeSource == WS_REAL_TIME_CLOCK);

        /* Disable the alarm */
        alarm.type = AT_NONE;
        request_resource(IO_SYSTEM);
        setClockAlarm(alarm);
        release_resource(IO_SYSTEM);
}
```

setdbase

Write Value to I/O Database

Syntax

```
#include <ctools.h>
void setdbase(unsigned type, unsigned address, int value);
```

Description

The **setdbase** function writes *value* to the I/O database. *type* specifies the method of addressing the database. *address* specifies the location in the database. The table below shows the valid address types and ranges

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

If the specified register is currently forced, the I/O database remains unchanged.

When writing to LINEAR digital addresses, *value* is a bit mask which writes data to 16 1-bit registers at once. If any of these 1-bit registers is currently forced, only the forced registers remain unchanged.

The I/O database is not modified when the controller is reset. It is a permanent storage area, which is maintained during power outages.

Refer to the **Functions Overview** section for more information.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

overrideDbase, setForceFlag

Example

```
#include <ctools.h>

void main(void)
{
	request_resource(IO_SYSTEM);

	setdbase(MODBUS, 40001, 102);

	/* Turn ON the first 16 coils */
```

```
        setdbase(LINEAR, START_COIL, 255);

        /* Write to a 16 bit register */
        setdbase(LINEAR, 3020, 240);

        /* Write to the 12th holding register */
        setdbase(LINEAR, START_HOLDING, 330);

        /* Write to the 12th holding register */
        setdbase(LINEAR, START_HOLDING, 330);

        release_resource(IO_SYSTEM);
}
```

setDTR

Control RS232 Port DTR Signal

Syntax

```
#include <ctools.h>
void setDTR(FILE *stream, unsigned state);
```

Description

The **setDTR** function sets the status of the DTR signal line for the communication port specified by *stream*. When *state* is SIGNAL_ON the DTR line is asserted. When *state* is SIGNAL_OFF the DTR line is de-asserted.

Notes

The DTR line follows the normal RS232 voltage levels for asserted and de-asserted states.

This function is only useful on RS232 ports. The function has no effect if the serial port is not an RS232 port.

setForceFlag

Set Force Flag State for a Register

Syntax

```
#include <ctools.h>
unsigned setForceFlag(unsigned type, unsigned address, unsigned
value);
```

Description

The **setForceFlag** function sets the force flag(s) for the specified database register(s) to *value*. *value* is either 1 or 0, or a 16-bit mask for LINEAR digital addresses. The valid range for *address* is determined by the database addressing *type*.

If the *address* or addressing *type* is not valid, force flags are left unchanged and FALSE is returned; otherwise TRUE is returned. The table below shows the valid address types and ranges.

Type	Address Ranges	Register Size
MODBUS	00001 to NUMCOIL 10001 to 10000 + NUMSTATUS 30001 to 30000 + NUMINPUT 40001 to 40000 + NUMHOLDING	1 bit 1 bit 16 bit 16 bit
LINEAR	0 to NUMLINEAR-1	16 bit

Notes

When a register's force flag is set, the value of the I/O database at that register is forced to its current value. This register's value can only be modified by using the **overrideDbase** function or the *Edit/Force Register dialog.* While forced this value can not be modified by the **setdbase** function, protocols, or Ladder Logic programs.

Force Flags are not modified when the controller is reset. Force Flags are in a permanent storage area, which is maintained during power outages.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

clearAllForcing, overrideDbase

Example

This program clears the force flag for register 40001 and sets the force flags for the 16 registers at linear address 302 (i.e. registers 10737 to 10752).

```
#include <ctools.h>
```

```
void main(void)
{
       request_resource(IO_SYSTEM);

       setForceFlag(MODBUS, 40001, 0);
       setForceFlag(LINEAR, 302, 255);

       release_resource(IO_SYSTEM);
}
```

setIOErrorIndication

Set I/O Module Error Indication

Syntax

```
Description#include <ctools.h>
void setIOErrorIndication(unsigned state);
```

The **setIOErrorIndication** function sets the I/O module error indication to the specified *state*. If set to TRUE, the I/O module communication status is reported in the controller status register and Status LED. If set to FALSE, the I/O module communication status is not reported.

Notes

Refer to the ***5203/4 System Manual*** or the ***SCADAPack System Manual*** for further information on the Status LED and Status Output.

See Also

getIOErrorIndication

setjiffy

Set the Jiffy Clock

Syntax

```
#include <ctools.h>
void setjiffy(unsigned long value);
```

Description

The **setjiffy** function sets the system jiffy clock. The jiffy clock increments every 1/60 second. The jiffy clock rolls over to 0 after 5183999. This is the number of 1/60-second intervals in a day.

Notes

The real time clock and the jiffy clock are not related. They may drift slightly with respect to each other over several days.

Use the jiffy clock to measure times with resolution better than the 1/10th resolution provided by timers.

See Also

interval

Example

See the example for the **jiffy** function.

setOutputsInStopMode

Set Outputs In Stop Mode

Syntax

```
#include <ctools.h>
void setOutputsInStopMode( unsigned doutsInStopMode, unsigned
aoutsInStopMode);
```

Description

The **setOutputsInStopMode** function sets the *doutsInStopMode* and *aoutsInStopMode* control flags to the specified state.

If *doutsInStopMode* is set to TRUE, then digital outputs are held at their last state when the Ladder Logic program is stopped. If *doutsInStopMode* is FALSE, then digital outputs are turned OFF when the Ladder Logic program is stopped.

If a*outsInStopMode* is TRUE, then analog outputs are held at their last value when the Ladder Logic program is stopped. If a*outsInStopMode* is FALSE, then analog outputs go to zero when the Ladder Logic program is stopped.

See Also

getOutputsInStopMode

Example

This program changes the output conditions to hold analog outputs at their last value when the Ladder Logic program is stopped.

#include <ctools.h>

```
void main(void)
{
	unsigned holdDoutsOnStop;
	unsigned holdAoutsOnStop;
	getOutputsInStopMode( &holdDoutsOnStop, &holdAoutsOnStop);
	holdAoutsOnStop = TRUE;
	setOutputsInStopMode( holdDoutsOnStop,  holdAoutsOnStop);
}
```

set_pid

Write PID Block Variable

Syntax

```
#include <ctools.h>
void set_pid(unsigned name, unsigned block, int value);
```

Description

The **set_pid** function assigns *value* to a PID control block variable. *name* needs to be specified by one of the variable name macros in **pid.h**. *block* needs to be in the range 0 to **PID_BLOCKS**-1.

Notes

See the ***Telepace PID Controllers Manual*** for a detailed description of PID control.

Values stored in PID blocks are not initialized when a program is run, and are guaranteed to retain their values during power failures and program loading. PID block variables must always be initialized by the user program.

The IO_SYSTEM resource must be requested before calling this function.

See Also

auto_pid, clear_pid

set_port

Set Serial Port Configuration

Syntax

```
#include <ctools.h>
void set_port(FILE *stream, struct pconfig *settings);
```

Description

The **set_port** function sets serial port communication parameters. *stream* needs to specify one of **com1**, **com2**, **com3** or **com4**. *settings* references a serial port configuration structure. Refer to the description of the pconfig structure for an explanation of the fields.

Notes

If the serial port settings are the same as the current settings, this function has no effect.

The serial port is reset when settings are changed. All data in the receive and transmit buffers are discarded.

To optimize performance, minimize the length of messages on com3 and com4. Examples of recommended uses for com3 and com4 are for local operator display terminals, and for programming and diagnostics using the Telepace program.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

get_port

Example

This code fragment changes the baud rate on com2 to 19200 baud.

```
#include <ctools.h>
struct pconfig settings;

get_port(com2, &settings);
settings.baud = BAUD19200;
request_resource(IO_SYSTEM);
set_port(com2, &settings);
release_resource(IO_SYSTEM);
```

This code fragment sets com2 to the same settings as com1.

```
#include <serial.h>
#include <primitiv.h>
struct pconfig settings;

request_resource(IO_SYSTEM);
```

```
set_port(com2, get_port(com1, &settings));
release_resource(IO_SYSTEM);
```

setPowerMode

Set Current Power Mode

Syntax

```
#include <ctools.h>
BOOLEAN setPowerMode(UCHAR cpuPower, UCHAR lan, UCHAR
usbPeripheral, UCHAR usbHost);
```

Description

The **setPowerMode** function returns TRUE if the new settings were successfully applied. The setPowerMode function allows for power savings to be realized by controlling the power to the LAN port, changing the clock speed, and individually controlling the host and peripheral USB power. The following table of macros summarizes the choices available.

Macro	Meaning
PM_CPU_FULL	The CPU is set to run at full speed
PM_CPU_REDUCED	The CPU is set to run at a reduced speed
PM_CPU_SLEEP	The CPU is set to sleep mode
PM_LAN_ENABLED	The LAN is enabled
PM_LAN_DISABLED	The LAN is disabled
PM_USB_PERIPHERAL_ENABLED	The USB peripheral port is enabled
PM_USB_PERIPHERAL_DISABLED	The USB peripheral port is disabled
PM_USB_HOST_ENABLED	The USB host port is enabled
PM_USB_HOST_DISABLED	The USB host port is disabled
PM_NO_CHANGE	The current value will be used

TRUE is returned if the requested change was made, otherwise FALSE is returned.

The application program may view the current power mode with the **getPowerMode** function.

See Also

getPowerMode, **setWakeSource**, **getWakeSource**

setProgramStatus

Set Program Status Flag

Syntax

#include <ctools.h>

void setProgramStatus(unsigned status);

Description

The **setProgramStatus** function sets the application program status flag. The status flag is set to **NEW_PROGRAM** when a cold boot of the controller is performed, or a program is downloaded to the controller from the program loader.

Notes

There are two pre-defined values for the flag. However the application program may make whatever use of the flag it sees fit.

NEW_PROGRAM	indicates the program is newly loaded.
PROGRAM_EXECUTED	indicates the program has been executed.

See Also

getProgramStatus

Example

See the example for **getProgramStatus**.

set_protocol

Set Communication Protocol Configuration

Syntax

```
#include <ctools.h>
int set_protocol(FILE *stream, struct prot_settings *settings);
```

Description

The **set_protocol** function sets protocol parameters. *stream* needs to specify one of **com1**, **com2**, **com3** or **com4**. *settings* references a protocol configuration structure. Refer to the description of the prot_settings structure for an explanation of the fields.

The **set_protocol** function returns **TRUE** if the settings were changed. It returns **FALSE** if there is an error in the settings or if the protocol does not start.

The IO_SYSTEM resource needs to be requested before calling this function.

Notes

Setting the protocol type to NO_PROTOCOL ends the protocol task and frees the stack resources allocated to it.

Be sure to add a call to modemNotification when writing a custom protocol.

See Also

get_protocol, start_protocol, modemNotification

Example

This code fragment changes the station number of the com2 protocol to 4.

```
#include <ctools.h>
struct prot_settings settings;

get_protocol(com2, &settings);
settings.station = 4;
request_resource(IO_SYSTEM);
set_protocol(com2, &settings);
release_resource(IO_SYSTEM);
```

setProtocolSettings

Set Protocol Extended Addressing Configuration

Syntax

```
#include <ctools.h>
BOOLEAN setProtocolSettings(
FILE * stream,
PROTOCOL_SETTINGS * settings
);
```

Description

The setProtocolSettings function sets protocol parameters for a serial port. This function supports extended addressing.

The function has two arguments: *stream* is one of com1, com2, com3 or com4; and *settings,* a pointer to a PROTOCOL_SETTINGS structure. Refer to the description of the structure for an explanation of the parameters.

The function returns **TRUE** if the settings were changed. It returns **FALSE** if the stream is not valid, or if the protocol does not start.

The IO_SYSTEM resource needs to be requested before calling this function.

Notes

Setting the protocol type to NO_PROTOCOL ends the protocol task and frees the stack resources allocated to it.

Be sure to add a call to modemNotification when writing a custom protocol.

Extended addressing is available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

See Also

getProtocolSettings, start_protocol, get_protocol, set_protocol, modemNotification

Example

This code fragment sets protocol parameters for the com2 serial port.

```
#include <ctools.h>
PROTOCOL_SETTINGS settings;

settings.type        = MODBUS_RTU;
settings.station     = 1234;
settings.priority    = 3;
settings.SFMessaging = FALSE;
settings.mode        = AM_extended;

request_resource(IO_SYSTEM);
setProtocolSettings(com2, &settings);
```

```
release_resource(IO_SYSTEM);
```

setProtocolSettingsEx

Sets extended protocol settings for a serial port.

Syntax

#include <ctools.h>

BOOLEAN setProtocolSettingsEx(

FILE * stream,

PROTOCOL_SETTINGS_EX * pSettings

);

Description

The setProtocolSettingsEx function sets protocol parameters for a serial port. This function supports extended addressing and Enron Modbus parameters.

The function has two arguments:

- stream specifies the serial port. It is one of com1, com2, com3 or com4.
- pSettings is a pointer to a PROTOCOL_SETTINGS_EX structure. Refer to the description of the structure for an explanation of the parameters.

The function returns TRUE if the settings were changed. It returns FALSE if the stream is not valid, or if the protocol does not start.

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

Setting the protocol type to NO_PROTOCOL ends the protocol task and frees the stack resources allocated to it.

Be sure to add a call to modemNotification when writing a custom protocol.

Extended addressing and the Enron Modbus station are available on the Modbus RTU and Modbus ASCII protocols only. See the *TeleBUS Protocols User Manual* for details.

See Also

getProtocolSettingsEx

Example

This code fragment sets protocol parameters for the com2 serial port.

```
#include <ctools.h>
PROTOCOL_SETTINGS_EX settings;

settings.type =         MODBUS_RTU;
settings.station =      1;
```

```
settings.priority =      3;
settings.SFMessaging =  FALSE;
settings.mode =          AM_standard;
settings.enronEnabled = TRUE;
settings.enronStation = 4;

request_resource(IO_SYSTEM);
setProtocolSettingsEx(com2, &settings);
release_resource(IO_SYSTEM);
```

setSFMapping

Control Translation Table Mapping

Syntax

```
#include <ctools.h>
void setSFMapping(unsigned flag);
```

Description

The s**etSFMapping** and **getSFMapping** functions no longer perform any useful function but are maintained as stubs for backward compatibility. Include the CNFG_StoreAndForward module in the Register Assignment to assign a store and forward table to the I/O database.

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

See Also

getSFMapping

setSFTranslation

Write Store and Forward Translation

Syntax

```
#include <ctools.h>
struct SFTranslationStatus setSFTranslation(unsigned index, struct
SFTranslation translation);
```

Description

The **setSFTranslation** function writes *translation* into the store and forward address translation table at the location specified by *index*. *translation* consists of two port and station address pairs. The function checks for invalid translations; if the translation is not valid it is not stored.

The function returns a SFTranslationStatus structure. It is described in the **Structures and Types** section. The *code* field of the structure is set to one of the following. If there is an error, the *index* field is set to the location of the translation that is not valid.

Result code	Meaning
SF_VALID	All translations are valid
SF_NO_TRANSLATION	The entry defines re-transmission of the same message on the same port
SF_PORT_OUT_OF_RANGE	One or both of the serial port indexes is not valid
SF_STATION_OUT_OF_RANGE	One or both of the stations is not valid
SF_ALREADY_DEFINED	The translation already exists in the table
SF_INDEX_OUT_OF_RANGE	The entry referenced by *index* does not exist in the table

Notes

The ***TeleBUS Protocols User Manual*** describes store and forward messaging mode.

Writing a translation with both stations set to station 256 can clear a translation in the table. Station 256 is not a valid station.

The protocol type and communication parameters may differ between serial ports. The store and forward messaging will translate the protocol messages.

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

getSFTranslation, clearSFTranslationTable, checkSFTranslationTable

Example

This program enables store and forward messaging on com1 and com2. Two entries are placed into the store and forward table.

The communication parameters and protocol type on com2 are different from com1.

```
#include <ctools.h>
void main(void)
{
       struct prot_settings settings;
       struct pconfig portset;
       struct SFTranslation translation;
       struct SFTranslationStatus status;

       request_resource(IO_SYSTEM);

       /* Set communication parameters for port 1 */
       portset.baud      = BAUD9600;
       portset.duplex    = FULL;
       portset.parity    = NONE;
       portset.data_bits = DATA8;
       portset.stop_bits = STOP1;
       portset.flow_rx   = DISABLE;
       portset.flow_tx   = DISABLE;
       portset.type      = RS232;
       portset.timeout   = 600;
       set_port(com1, &portset);

       /* Set communication parameters for port 2 */
       portset.baud      = BAUD1200;
       portset.duplex    = HALF;
       portset.parity    = NONE;
       portset.data_bits = DATA8;
       portset.stop_bits = STOP1;
       portset.flow_rx   = DISABLE;
       portset.flow_tx   = DISABLE;
       portset.type      = RS232;
       portset.timeout   = 600;
       set_port(com2, &portset);

       /* Set up the translation table */
       clearSFTranslationTable();

       translation.portA    = portIndex(com1);
       translation.stationA = 2;
       translation.portB    = portIndex(com2);
       translation.stationB = 3;
       setSFTranslation(0, translation);

       translation.portA    = portIndex(com1);
       translation.stationA = 4;
       translation.portB    = portIndex(com2);
       translation.stationB = 5;
       setSFTranslation(1, translation);
```

```
        /* Enable store and forward messaging */
        settings.type        = MODBUS_RTU;
        settings.station     = 1;
        settings.priority    = 3;
        settings.SFMessaging = TRUE;
        set_protocol(com1, &settings);

        settings.type        = MODBUS_ASCII;
        settings.station     = 1;
        settings.priority    = 3;
        settings.SFMessaging = TRUE;
        set_protocol(com2, &settings);

        release_resource(IO_SYSTEM);

        /* Check if everything is correct */
        status = checkSFTranslationTable();
        if (status.code != SF_VALID)
        {
                /* Blink the error code on the status LED */
                setStatus(status.code);
        }
        else
        {
                setStatus(0);
        }

        while (TRUE)
        {
                /* main loop of application program */
        }
}
```

setStatus

Set Controller Status Code

Syntax

```
#include <ctools.h>
void setStatus(unsigned code);
```

Description

The **setStatus** function sets the controller status code. When the status code is non-zero, the STAT LED blinks a binary sequence corresponding to the code. If *code* is zero, the STAT LED turns off.

Notes

The status output opens if *code* is non-zero. Refer to the ***System Hardware Manual*** for more information.

The binary sequence consists of short and long flashes of the error LED. A short flash of 1/10th of a second indicates a binary zero. A binary one is indicated by a longer flash of approximately 1/2 of a second. The least significant digit is output first. As few bits as possible are displayed –leading zeros are ignored. There is a two second delay between repetitions.

The Register Assignment uses bits 0 and 1 of the status code. It is recommended that the **setStatusBit** function be used instead of **setStatus** to prevent modification of these bits.

See Also

setStatusBit, clearStatusBit, getStatusBit

setStatusBit

Set Bits in Controller Status Code

Syntax

```
#include <ctools.h>
unsigned setStatusBit(unsigned bitMask);
```

Description

The **setStatusBit** function sets the bits indicated by *bitMask* in the controller status code. When the status code is non-zero, the STAT LED blinks a binary sequence corresponding to the code. If *code* is zero, the STAT LED turns off.

The function returns the value of the status register.

Notes

The status output opens if *code* is non-zero. Refer to the ***System Hardware Manual*** for more information.

The binary sequence consists of short and long flashes of the STAT LED. A short flash of 1/10th of a second indicates a binary zero. A binary one is indicated by a longer flash of approximately 1/2 of a second. The least significant digit is output first. As few bits as possible are displayed – all leading zeros are ignored. There is a two second delay between repetitions.

The Register Assignment uses bits 0 and 1 of the status code.

See Also

clearStatusBit, clearStatusBit, getStatusBit

settimer

Set a Timer

Syntax

```
#include <ctools.h>
void settimer(unsigned timer, unsigned value);
```

Description

The **settimer** function loads *value* into timer *specified by timer*. The timer counts down at the timer interval frequency.

The **settimer** function can reset a timer before it has finished counting down.

Notes

The **settimer** function cancels delayed digital I/O actions started with the **timeout**, **pulse** and **pulse_train** functions..

See Also

interval

Example

This code fragment sets timer 8 for 10 seconds, using an interval of 0.5 seconds.

```
interval(8, 5);     /* interval = 1/2 second */
settimer(8, 20);    /* 10 second timer */

This code fragment sets timer 9 for 60 seconds using an interval
of 1.0 seconds.
interval(9, 10);    /* interval = 1 second */
settimer(9, 60);    /* 60 second timer */
```

setWakeSource

Sets Conditions for Waking from Sleep Mode

Syntax

```
#include <ctools.h>
void setWakeSource(unsigned enableMask);
```

Description

The setWakeSource routine enables and disables sources that will wake up the processor. It enables all sources specified by *enableMask*. All other sources are disabled.

Valid wake up sources are listed below. Multiple sources may be ORed together.

- WS_NONE
- WS_ALL
- WS_REAL_TIME_CLOCK
- WS_INTERRUPT_INPUT
- WS_LED_POWER_SWITCH
- WS_COUNTER_0_OVERFLOW
- WS_COUNTER_1_OVERFLOW
- WS_COUNTER_2_OVERFLOW

Notes

Specifying WS_NONE as the wake up source will prevent the controller from waking, except by a power on reset.

See Also

getWakeSource, sleep

Example

The code fragments below show how to enable and disable wake up sources.

```
/* Wake up on all sources */
setWakeSource(WS_ALL);

/* Enable wake up on real time clock only */
setWakeSource(WS_REAL_TINE_CLOCK);
```

signal_event

Signal Occurrence of Event

Syntax

```
#include <ctools.h>
void signal_event(int event_number);
```

Description

The **signal_event** function signals that the *event_number* event has occurred.

If there are tasks waiting for the event, the highest priority task is made ready to execute. Otherwise the event flag is incremented. Up to 255 occurrences of an event will be recorded. The current task is blocked of there is a higher priority task waiting for the event.

Notes

Refer to the **Real Time Operating System** section for more information on events.

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in ctools.h are not valid events for use in an application program.

See Also

wait_event

Example

This program creates a task to wait for an event, then signals the event.

```
#include <ctools.h>

void task1(void)
{
       while(TRUE)
       {
              wait_event(20);
              printf("Event 20 occurred\r\n");
       }
}

void main(void)
{
       create_task(task1, 3, APPLICATION, 4);

       while(TRUE)
       {
              /* body of main task loop */
/* The body of this main task is intended solely for signaling the
event waited for by task1. Normally main would be busy with more
```

```
important things to do otherwise the code in task1 could be
executed within main's wait loop */

                settimer(0, 10);            /* 1 second interval */
                while (timer(0))            /* wait for 1 s */
                {
                        /* Allow other tasks to execute */
                        release_processor();
                }
                signal_event(20);
        }
}
```

sleep

Suspend Controller Operation

Syntax

```
#include <ctools.h>
unsigned sleep(void);
```

Description

The **sleep** function puts the controller into a sleep mode. Sleep mode reduces the power consumption to a minimum by halting the microprocessor clock and shutting down the power supply. All programs halt until the controller resumes execution. All output points turn off while the controller is in sleep mode.

The controller resumes execution under the conditions shown in the table below. The application program may disable some wake up conditions. If a wake up condition is disabled the controller will not resume execution when the condition occurs. The table below shows the effect of disabling the various wake up conditions. All wake up conditions will be enabled by default. Refer to the description of the **setWakeSource** function for details.

Condition	Wake Up Effects	Disable Allowed	Disable Effect
Hardware Reset	Application programs execute from start of program.	No	Not applicable.
External Interrupt	Program execution continues from point sleep function was executed.	Yes	Interrupt input ignored
Real Time Clock Alarm	Program execution continues from point sleep function was executed.	Yes	Alarm ignored
LED Power Button	Program execution continues from point sleep function was executed.	Yes	LED power button ignored
Hardware Counter Rollover	Software portion of counter is incremented. Program execution continues from point sleep function was executed.	Yes	Software portion of counter is incremented. Controller returns to sleep mode.

The **sleep** function returns a wake up code indicating which condition caused the controller to resume execution.

Return Code	**Condition**
WS_REAL_TIME_CLOCK	real time clock alarm
WS_INTERRUPT_INPUT	rising edge of interrupt input
WS_LED_POWER_SWITCH	LED Power switch pushed
WS_COUNTER_0_OVERFLOW	roll over of low word of counter 0 (every 65536 transitions)
WS_COUNTER_1_OVERFLOW	roll over of low word of counter 1 (every 65536 transitions)
WS_COUNTER_2_OVERFLOW	roll over of low word of counter 2 (every 65536 transitions)

Notes

The IO_SYSTEM resource needs to be requested before calling this function.

See Also

setclock, alarmIn, setWakeSource, getWakeSource

Example

See the examples for the **setClockAlarm** and **alarmIn** functions.

start_protocol

Enable Protocol Task

Syntax

```
#include <ctools.h>
int start_protocol(FILE *stream);
```

Description

The **start_protocol** function enables a protocol task on the port specified by *stream*. The protocol configuration settings stored in memory are used.

The **start_protocol** function returns **TRUE** if the protocol started and **FALSE** if there was an error.

Notes

The **start_protocol** function is used by the system start up routine. Application programs should use the **set_protocol** function to control protocol operation.

See Also

get_protocol, set_protocol

startup_task

Identify Start Up Task

Syntax

```
#include <ctools.h>
void *startup_task(void);
```

Description

The **startup_task** function returns the address of the system or application start up task.

Notes

This function is used by the reset routine. It is normally not used in an application program.

startTimedEvent

Enable Signaling of a Regular Event

Syntax

```
#include <ctools.h>
unsigned startTimedEvent(unsigned event, unsigned interval);
```

Description

The **startTimedEvent** function causes the specified *event* to be signaled at the specified *interval*. *interval* is measured in multiples of 0.1 seconds. The task that is to receive the events should use the **wait_event** or **poll_event** functions to detect the event.

The function returns TRUE if the event can be signaled. If interval is 0 or if the event number is not valid, the function returns FALSE and no change is made to the event signaling (a previously enabled event will not be changed).

Notes

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in primitiv.h are not valid events for use in an application program.

The application program should stop the signaling of timed events when the task which waits for the events is ended. If the event signaling is not stopped, events will continue to build up in the queue until a function waits for them. The example below shows a simple method using the **installExitHandler** function.

See Also

endTimedEvent, signal_event, wait_event

Example

The program prints the time every 10 seconds.

```
#include <string.h>
#include <ctools.h>

#define TIME_TO_PRINT      15

/* --------------------------------------------
   The shutdown function stops the signalling
   of TIME_TO_PRINT events.
   -------------------------------------------- */
void shutdown(void)
{
       endTimedEvent(TIME_TO_PRINT);
}

/* --------------------------------------------
   The main function sets up signalling of
   a timed event, then waits for that event.
```

```
   The time is printed each time the event
   occurs.
   ------------------------------------------- */
void main(void)
{
       struct prot_settings settings;
       struct clock now;
       TASKINFO taskStatus;

       /* Disable the protocol on serial port 1 */
       settings.type = NO_PROTOCOL;
       settings.station = 1;
       settings.priority = 3;
       settings.SFMessaging = FALSE;
       request_resource(IO_SYSTEM);
       set_protocol(com1, &settings);
       release_resource(IO_SYSTEM);

       /* set up task exit handler to stop
          signalling of events when this task ends */
       taskStatus = getTaskInfo(0);
       installExitHandler(taskStatus.taskID, shutdown);

       /* start timed event */
       startTimedEvent(TIME_TO_PRINT, 100);

       while (TRUE)
       {
              wait_event(TIME_TO_PRINT);
              request_resource(IO_SYSTEM);
              now = getclock();
              release_resource(IO_SYSTEM);
              fprintf(com1, "Time %02u:%02u:%02u\r\n", now.hour,
now.minute, now.second);
       }
}
```

timeout

Delayed Digital Output

Syntax

```
#include <ctools.h>
void timeout(unsigned channel, unsigned bit, unsigned timer,
unsigned delay);
```

Description

The **timeout** function initiates a delayed control action on a digital output. The output changes state when the delay expires.

- *channel* specifies the digital output channel.
- *bit* specifies the output point within *channel*.
- *timer* specifies the timer used to measure the delay. It must be in the range 0 to 31.
- *delay* specifies the delay in timer ticks. The **interval** function sets the length of a timer tick.

If an error occurs, the current task's error code is set as follows:

TIMER_BADTIMER	if the timer number is invalid
TIMER_BADADDR	if the digital channel or bit is invalid

Notes

To cancel a timeout, set the timer to zero.

Use the **pulse** function to generate a repeating square wave.

The **timeout** function may start a new timeout sequence before the previous one completes. In this case, the previous timeout sequence is canceled and the new one begins.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioWrite8Dout** directly.

See Also

interval, ioWrite8Dout, turnoff, turnon, settimer, pulse

timeoutCancel

Cancel Timeout Notification Function

Syntax

```
#include <ctools.h>
unsigned timeoutCancel(unsigned timeoutID);
```

Description

This function cancels a timeout notification that was requested with the timeoutRequest function. No notification will be sent. The envelope provided when the request was made is de-allocated.

The function has one parameter: the ID of the timeout request. This is the value returned by the timeoutRequest function.

The function returns TRUE if the request was cancelled and FALSE if the timeout ID is not currently active.

Notes

The function will return FALSE if the timeout notification has already been made. In this case the envelope will not be de-allocated as it has already been given to the destination task. That task is responsible for de-allocating the envelope.

This function cannot be called from a task exit handler. See installExitHandler function for details of exit handlers.

See Also

timeoutRequest

Example

See the example for the timeoutRequest function.

timeoutRequest

Request Timeout Notification Function

Syntax

```
#include <ctools.h>
unsigned timeoutRequest(unsigned delay, envelope * pEnvelope);
```

Description

This function requests a timeout notification. A message is sent to the task specified in the envelope after the specified delay.

A task receives the message using the receive_message or poll_message function. The envelope received by the receiving task has the following characteristics.

- The source field is set to the task ID of the task that called timeoutRequest.
- The message type field is set to MSG_TIMEOUT.
- The message data is set to the timeout ID.

The function has two parameters: the length of time in tenths of a second before the timeout occurs, and a pointer to an envelope. The resolution of the delay is –0.1/+0 seconds. The notification message is sent delay-1 to delay tenths of a second after the function call.

The function returns the ID of the timeout request. This can be used to identify and cancel the timeout. The timeout ID changes with each call to the function. Although the ID will eventually repeat, it is sufficiently unique to allow the timeout notification to be identified. This can be useful in identifying notifications received by a task and matching them with requests.

Notes

Do not de-allocate the envelope passed to timeoutRequest in the calling function. After a call to timeoutRequest either use timeoutCancel to free the envelope if the timeout has not occurred yet, or call deallocate_envelope in the destination task after the envelope has been delivered.

The timeout may be cancelled using the timeoutCancel function.

The task that receives the notification message needs to de-allocate the envelope after receiving it.

No checking is done on the task ID. The caller needs to ensure it is valid.

If the delay is zero, the message is sent immediately, provided an envelope is available.

This function cannot be called from a task exit handler. See installExitHandler function for details of exit handlers.

See Also

timeoutCancel

Example

This example shows a task that acts on messages received from other tasks and when a timeout occurs. The task waits for a message for up to 10 seconds. If it does not receive one, it proceeds with other processing anyway.

The task shows how to deal with notifications from older timeout requests. These occur when the notification was send before the timeout was cancelled. The task ignores timeout notifications that don't match the last timeout request.

```
#include <mriext.h>
#include <ctools.h>

void aTask(void)
{
envelope * pEnvelope;
TASKINFO thisTask;
unsigned timeoutID;
unsigned done;

/* get the task ID for this task */
thisTask = getTaskInfo(0);

while (TRUE)
       {
       /* allocate an envelope and address it to this task */
       pEnvelope = allocate_envelope();
       pEnvelope->destination = thisTask.taskID;

       /* request a timeout in 10 seconds */
       timeoutID = timeoutRequest(100, pEnvelope);

       done = FALSE;
       while (!done)
              {
              /* wait for a message or a timeout */
              pEnvelope = receive_message();

              /* determine the message type */
              if (pEnvelope->type == MSG_TIMEOUT)
                     {
                     /* does it match the last request? */
                     if (pEnvelope->data == timeoutID)
                            {
                            /* accept the timeout */
                            done = TRUE;
                            }
                     }
              else
                     {
                     /* cancel the timeout */
                     timeoutCancel(timeoutID);
```

```
                        done = TRUE;

                        /* process message from other task here */
                        }

                /* return the envelope to the RTOS */
                deallocate_envelope(pEnvelope);
                }

        /* proceed with rest of task's actions here */
        }
}
```

timer

Read a Timer

Syntax

#include <ctools.h>

unsigned timer(unsigned *timer*);

Description

The **timer** function returns the time remaining in *timer*. *timer* needs to be in the range 0 to 31. A zero value means that the timer has finished counting down.

If the timer number is invalid, the function returns 0 and the task's error code is set to **TIMER_BADTIMER**.

See Also

interval, settimer, timeout, read_timer_info, pulse

Example

This code fragment sets a timer, then displays the time remaining until it reaches 0.

```
#include <ctools.h>

interval(0, 1);
settimer(0, 10);
while (timer(0))
      printf("Time %d\r\n", timer(0));
```

turnoff

Turn Off a Digital Output

Syntax

```
#include <ctools.h>
int turnoff(unsigned channel, unsigned bit);
```

Description

The **turnoff** function turns off the digital output specified by *channel* and *bit*.

The **turnoff** function returns the value written to the channel if successful. If *channel* or *bit* is invalid, it returns –1.

Notes

The **turnoff** function has no effect if the specified point is configured as a digital input.

The state of the physical output is modified by the values in the I/O form, disable, and force status tables.

Multiple bits in the same channel can be set with the **dout** function.

Use offsets from the symbolic constants DIN_START, DIN_END, DOUT_START and DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioWrite8Dout** directly.

See Also

ioWrite8Dout, turnon

turnon

Turn On a Digital Output

Syntax

```
#include <ctools.h>
int turnon(unsigned channel, unsigned bit);
```

Description

The **turnon** function turns on the digital output specified by *channel* and *bit*.

The **turnon** function returns the value written to the channel if successful. If *channel* or *bit* is invalid, it returns –1.

Notes

The **turnon** function has no effect if the specified point is configured as a digital input.

The state of the physical output is modified by the values in the I/O form, disable, and force status tables.

Multiple bits in the same channel can be set with the **dout** function.

Use offsets from the symbolic constants DIN_START, DIN_END, DOUT_START and DOUT_END to reference digital channels. The constants make programs more portable and protect against future changes to the digital I/O channel numbering.

The IO_SYSTEM resource needs to be requested before calling this function.

This function is provided for backward compatibility. It cannot access all 5000 I/O modules. It is recommended that this function not be used in new programs. Instead use Register Assignment or call the I/O driver **ioWrite8Dout** directly.

See Also

ioWrite8Dout, turnoff

wait_event

Wait for an Event

Syntax

```
#include <ctools.h>
void wait_event(int event);
```

Description

The **wait_event** function tests if an event has occurred. If the event has occurred, the event counter is decrements and the function returns. If the event has not occurred, the task is blocked until it does occur.

Notes

Refer to the **Real Time Operating System** section for more information on events.

Valid events are numbered 0 to RTOS_EVENTS - 1. Any events defined in primitiv.h are not valid events for use in an application program.

See Also

signal_event, startTimedEvent

Example

See the example for the **signal_event** function.

wd_auto

Automatic Watchdog Timer Mode

Syntax

```
#include <ctools.h>
void wd_auto(void);
```

Description

The **wd_auto** function gives control of the watchdog timer to the operating system. The timer is automatically updated by the system.

Notes

Refer to the **Functions Overview** section for more information.

See Also

wd_manual, wd_pulse

Example

See the example for the **wd_manual** function

wd_manual

Manual Watchdog Timer Mode

Syntax

```
#include <ctools.h>
void wd_manual(void);
```

Description

The **wd_manual** function takes control of the watchdog timer.

Notes

The application program needs to retrigger the watchdog timer at least every 0.5 seconds using the **wd_pulse** function, to prevent an controller reset.

Refer to the **Functions Overview** section for more information.

See Also

wd_auto, wd_pulse

Example

This program takes control of the watchdog timer for a section of code, then returns it to the control of the operating system.

```
#include <ctools.h>

void main(void)
{
	wd_manual();
	wd_pulse();
	/* ... code executing in less than 0.5 s */
	wd_pulse();
	/* ... code executing in less than 0.5 s */
	wd_auto()
	/* ... as much code as you wish */
}
```

wd_pulse

Retrigger Watchdog Timer

Syntax

```
#include <ctools.h>
void wd_pulse(void);
```

Description

The **wd_pulse** function retriggers the watchdog timer.

Notes

The **wd_pulse** function must execute at least every 0.5 seconds, to prevent an controller reset, if the **wd_manual** function has been executed.

Refer to the **Functions Overview** section for more information.

See Also

wd_auto, wd_manual

Example

See the example for the **wd_manual** function

Telepace C Tools Macro Definitions

A

Macro	Definition
AB	Specifies Allan-Bradley database addressing.
AB_PARSER	System resource: DF1 protocol message parser.
AB_FULL_BCC	Specifies the DF1 Full Duplex protocol emulation for the serial port. (BCC checksum)
AB_FULL_CRC	Specifies the DF1 Full Duplex protocol emulation for the serial port. (CRC checksum)
AB_HALF_BCC	Specifies the DF1 Half Duplex protocol emulation for the serial port. (BCC checksum)
AB_HALF_CRC	Specifies the DF1 Half Duplex protocol emulation for the serial port. (CRC checksum)
AB_PROTOCOL	DF1 protocol firmware option
AD_BATTERY	Internal AD channel connected to lithium battery
AD_THERMISTOR	Internal AD channel connected to thermistor
ADDITIVE	Additive checksum
AIN_END	Number of last analog input channel.
AIN_START	Number of first analog input channel.
AIO_BADCHAN	Error code: bad analog input channel specified.
AIO_SUPPORTED	If defined indicates analog I/O supported.
AIO_TIMEOUT	Error code: input device did not respond.
AO	Variable name: alarm output address
AOUT_END	Number of last analog output channel.
AOUT_START	Number of first analog output channel.
APPLICATION	Specifies an application type task. All application tasks are terminated by the end_application function.

Macro	Definition
AT_ABSOLUTE	Specifies a fixed time of day alarm.
AT_NONE	Disables alarms

B

Macro	Definition
BACKGROUND	System event: background I/O requested. The background I/O task uses this event. It should not be used in an application program.
BASE_TYPE_MASK	Controller type bit mask
BAUD110	Specifies 110-baud port speed.
BAUD115200	Specifies 115200-baud port speed.
BAUD1200	Specifies 1200-baud port speed.
BAUD150	Specifies 150-baud port speed.
BAUD19200	Specifies 19200-baud port speed.
BAUD2400	Specifies 2400-baud port speed.
BAUD300	Specifies 300-baud port speed.
BAUD38400	Specifies 38400-baud port speed.
BAUD4800	Specifies 4800-baud port speed.
BAUD57600	Specifies 57600-baud port speed.
BAUD600	Specifies 600-baud port speed.
BAUD75	Specifies 75-baud port speed.
BAUD9600	Specifies 9600-baud port speed.
BYTE_EOR	Byte-wise exclusive OR checksum

C

Macro	Definition
CA	Variable name: cascade setpoint source
CLASS0_FLAG	specifies a flag for enabling DNP Class 0 data
CLASS1_FLAG	specifies a flag for enabling DNP Class 1 data
CLASS2_FLAG	specifies a flag for enabling DNP Class 2 data
CLASS3_FLAG	specifies a flag for enabling DNP Class 3 data

Macro	Definition
CLOSED	Specifies switch is in closed position
COLD_BOOT	Cold-boot switch depressed when CPU was reset.
com1	Points to a file object for the com1 serial port.
COM1_RCVR	System event: indicates activity on com1 receiver. The meaning depends on the character handler installed.
com2	Points to a file object for the com2 serial port.
COM2_RCVR	System event: indicates activity on com2 receiver. The meaning depends on the character handler installed.
com3	Points to a file object for the com3 serial port.
COM3_RCVR	System event: indicates activity on com3 receiver. The meaning depends on the character handler installed.
com4	Points to a file object for the com4serial port.
COM4_RCVR	System event: indicates activity on com4 receiver. The meaning depends on the character handler installed.
COUNTER_CHANNELS	Specifies number of 5000 I/O counter input channels
COUNTER_END	Number of last counter input channel
COUNTER_START	Number of first counter input channel
COUNTER_SUPPORTED	If defined indicates counter I/O hardware supported.
CPU_CLOCK_RATE	Frequency of the system clock in cycles per second
CR	Variable name: control register
CRC_16	CRC-16 type CRC checksum (reverse algorithm)
CRC_CCITT	CCITT type CRC checksum (reverse algorithm)

D

Macro	Definition
DATA_SIZE	Maximum length of the HART command or

Macro	Definition
	response field.
DATA7	Specifies 7 bit world length.
DATA8	Specifies 8 bit word length.
DB	Variable name: deadband
DB_BADSIZE	Error code: out of range address specified
DB_BADTYPE	Error code: bad database addressing type specified
DB_OK	Error code: no error occurred
DCA_ADD	Add the ID to the configuration registers.
DCA_REMOVE	Remove the ID from the configuration registers.
DCAT_C	Device configuration application type is a C application
DCAT_LOGIC1	Device configuration application type is the first logic application
DCAT_LOGIC2	Device configuration application type is the second logic application
DE_BadConfig	The modem configuration structure contains an error
DE_BusyLine	The phone number called was busy
DE_CallAborted	A call in progress was aborted by the user
DE_CarrierLost	The connection to the remote site was lost (modem reported NO CARRIER). Carrier is lost for a time exceeding the S10 setting in the modem. Phone lines with call waiting are very susceptible to this condition.
DE_FailedToConnect	The modem could not connect to the remote site
DE_InitError	Modem initialization failed (the modem may be turned off)
DE_NoDialTone	Modem did not detect a dial tone or the S6 setting in the modem is too short.
DE_NoError	No error has occurred
DE_NoModem	The serial port is not configured as a modem (port type must be RS232_MODEM). Or no modem is connected to the controller serial port.
DE_NotInControl	The serial port is in use by another modem function or has answered an incoming call.
DIN_END	Number of last regular digital input channel.
DIN_START	Number of first regular digital input channel

Macro	Definition
DIO_SUPPORTED	If defined indicates digital I/O hardware supported.
DISABLE	Specifies flow control is disabled.
DNP	Specifies the DNP protocol for the serial port
DO	Variable name: decrease output
DOUT_END	Number of last regular digital output channel.
DOUT_START	Number of first regular digital output channel
DS_Calling	The controller is making a connection to a remote controller
DS_Connected	The controller is connected to a remote controller
DS_Inactive	The serial port is not in use by a modem
DS_Terminating	The controller is ending a connection to a remote controller.
DUTY_CYCLE	Specifies timer is generating square wave output.
DYNAMIC_MEMORY	System resource: all memory allocation functions such as malloc, alloc, and zalloc.

E

Macro	Definition
EEPROM_EVERY	EEPROM section loaded to RAM on every CPU reboot
EEPROM_RUN	EEPROM section loaded to RAM on RUN type boots only.
EEPROM_SUPPORTED	If defined, indicates that there is an EEPROM in the controller.
ENABLE	Specifies flow control is enabled.
ER	Variable name: error
EVEN	Specifies even parity.
EX	Variable name: automatic execution period
EXTENDED_DIN_END	Number of last extended digital input channel.
EXTENDED_DIN_START	Number of first extended digital input channel
EXTENDED_DOUT_END	Number of last extended digital output channel.

Macro	Definition
EXTENDED_DOUT_START	Number of first extended digital output channel

F

Macro	Definition
FOPEN_MAX	Redefinition of macro from stdio.h
FORCE_MULTIPLE_COILS	Modbus function code
FORCE_SINGLE_COIL	Modbus function code
FOXCOM_MESSAGE_RECEIVED	This event is used when a Foxcom message is received. An application program cannot use this event.
FOXCOM_STARTED	This event is used when Foxcom communication has been established with a sensor. An application program cannot use this event.
FS	Variable name: full scale output limit
FULL	Specifies full duplex.

G

Macro	Definition
GA	Variable name: gain
GASFLOW	Gas Flow calculation firmware option
GFC_4202	SCADAPack 4202 DR controller
GFC_4202DS	SCADAPack 4202 DS controller

H

Macro	Definition
HALF	Specifies half duplex.
HI	Variable name: high alarm setpoint

I

Macro	Definition
IB	Variable name: input bias
IH	Variable name: inhibit execution address
IN	Variable name: integrated error
IO	Variable name: increase output
IO_SYSTEM	System resource for all I/O hardware functions.
IP	Variable name: input source

L

Macro	Definition
LED_OFF	Specifies LED is to be turned off.
LED_ON	Specifies LED is to be turned on.
LINEAR	Specifies linear database addressing.
LO	Variable name: low alarm setpoint
LOAD_MULTIPLE_REGISTERS	Modbus function code
LOAD_SINGLE_REGISTER	Modbus function code
LOCAL_COUNTERS	Number of 5203/4 counter inputs

M

Macro	Definition
MAX_PRIORITY	The maximum task priority.
MM_BAD_ADDRESS	Master message status: invalid database address
MM_BAD_FUNCTION	Master message status: invalid function code
MM_BAD_LENGTH	Master message status: invalid message length
MM_BAD_SLAVE	Master message status: invalid slave station address
MM_NO_MESSAGE	Master message status: no message was sent.
MM_PROTOCOL_NOT_SUPPORTED	Master message status: selected protocol is not supported.
MM_RECEIVED	Master message status: response received.

Macro	Definition
MM_RECEIVED_BAD_LENGTH	Master message status: response received with the incorrect amount of data.
MM_SENT	Master message status: message was sent.
MODBUS	Specifies Modbus database addressing.
MM_EOT	Master message status: DF1 slave response was an EOT message
MM_WRONG_RSP	Master message status: DF1slave response did not match command sent.
MM_CMD_ACKED	Master message status: DF1half duplex command has been acknowledged by slave – Master may now send poll command.
MM_EXCEPTION_ADDRESS	Master message status: Modbus slave returned an address exception.
MM_EXCEPTION_DEVICE_BUSY	Master message status: Modbus slave returned a Device Busy exception.
MM_EXCEPTION_DEVICE_FAILURE	Master message status: Modbus slave returned a Device Failure exception
MM_EXCEPTION_FUNCTION	Master message status: Modbus slave returned a function exception.
MM_EXCEPTION_VALUE	Master message status: Modbus slave returned a value exception.
MODBUS_ASCII	Specifies the Modbus ASCII protocol emulation for the serial port.
MODBUS_PARSER	System resource: Modbus protocol message parser.
MODBUS_RTU	Specifies the Modbus RTU protocol emulation for the serial port.
MODEM_CMD_MAX_LEN	Maximum length of the modem initialization command string
MODEM_MSG	System event: new modem message generated.
MSG_DATA	Specifies the data field in an envelope contains a data value.
MSG_POINTER	Specifies the data field in an envelope contains a pointer.

N

Macro	Definition
NEVER	System event: this event will never occur.
NEW_PROGRAM	Application program is newly loaded.
NO_ERROR	Error code: indicates no error has occurred.
NO_PROTOCOL	Specifies no communication protocol for the serial port.
NONE	Specifies no parity.
NORMAL	Specifies normal count down timer.
NORMAL	Specifies normal count down timer.
NOTYPE	Specifies serial port type is not known.
NUMAB	Number of registers in the Allan-Bradley database.
NUMCOIL	Number of registers in the Modbus coil section.
NUMHOLDING	Number of registers in the Modbus holding register section.
NUMINPUT	Number of registers in the Modbus input register section.
NUMLINEAR	Number of registers in the linear database.
NUMSTATUS	Number of registers in the Modbus status section.

O

Macro	Definition
OB	Variable name: output bias
ODD	Specifies odd parity.
OB	Variable name: output bias
OP	Variable name: output
OPEN	Specifies switch is in open position

P

Macro	Definition
PC_FLOW_RX_RECEIVE_STOP	Receiver disabled after receipt of a message.
PC_FLOW_RX_XON_XOFF	Receiver Xon/Xoff flow control.
PC_FLOW_TX_IGNORE_CTS	Transmitter flow control ignores CTS.
PC_FLOW_TX_XON_XOFF	Transmitter Xon/Xoff flow control.

Macro	Definition
PC_PROTOCOL_RTU_FRAMING	Modbus RTU framing.
PID_ALARM	Control register mask: alarms enabled
PID_ALARM_ABS	Control register mask: absolute alarms
PID_ALARM_ACK	Status register mask: alarm acknowledged
PID_ALARM_DEV	Control register mask: deviation alarms
PID_ALARM_ONLY	Control register mask: alarm only block
PID_ALARM_RATE	Control register mask: rate alarms
PID_ANALOG_IP	Control register mask: analog input
PID_ANALOG_OP	Control register mask: analog output
PID_BAD_BLOCK	Return code: bad block number specified.
PID_BAD_IO_IP	Status register mask: I/O failure on block input
PID_BAD_IO_OP	Status register mask: I/O failure on block output
PID_BLOCK_IP	Control register mask: input from output of another block
PID_BLOCKS	Number of PID blocks.
PID_CLAMP_FULL	Status register mask: output is clamped at full scale
PID_CLAMP_ZERO	Status register mask: output is clamped at zero scale
PID_ER_SQR	Control register mask: take square root of error
PID_HI_ALARM	Status register mask: high alarm detected
PID_INHIBIT	Status register mask: external inhibit input is on
PID_LO_ALARM	Status register mask: low alarm detected
PID_MANUAL	Status register mask: block is in manual mode
PID_MODE_AUTO	Control register mask: automatic mode
PID_MODE_MANUAL	Control register mask: manual mode
PID_MOTOR_OP	Control register mask: motor pulse duration output
PID_NO_ALARM	Control register mask: alarms disabled
PID_NO_ER_SQR	Control register mask: normal error
PID_NO_IP	Control register mask: no input (other than IP)
PID_NO_OP	Control register mask: no output
PID_NO_PV_SQR	Control register mask: normal PV

Macro	Definition
PID_NO_SP_TRACK	Control register mask: setpoint tracking disabled
PID_OK	Return code: operation completed successfully.
PID_OUT_DB	Status register mask: PID controller outside of deadband
PID_PID	Control register mask: PID control block
PID_PULSE_OP	Control register mask: pulse duration output
PID_PV_SQR	Control register mask: take square root of PV
PID_RATE_CLAMP	Status register mask: rate gain clamed at maximum
PID_RATIO_BIAS	Control register mask: ratio/bias control block
PID_RUNNING	Status register mask: block is executing
PID_SP_CASCADE	Control register mask: cascade setpoint
PID_SP_NORMAL	Control register mask: setpoint stored in SP
PID_SP_TRACK	Control register mask: setpoint tracking enabled
PE	Variable name: period
PHONE_NUM_MAX_LEN	Maximum length of the phone number string
PROGRAM_EXECUTED	Application program has been executed.
PULSE_TRAIN	Specifies timer is generating pulse train output.
PV	Variable name: process value
PM_CPU_FULL_CLOCK	The CPU is set to run at full speed
PM_CPU_REDUCED_CLOCK	The CPU is set to run at a reduced speed
PM_CPU_SLEEP	The CPU is set to sleep mode
PM_LAN_ENABLED	The LAN is enabled
PM_LAN_DISABLED	The LAN is disabled
PM_USB_PERIPHERAL_ENABLED	The USB peripheral port is enabled
PM_USB_PERIPHERAL_DISABLED	The USB peripheral port is disabled
PM_USB_HOST_ENABLED	The USB host port is enabled
PM_USB_HOST_DISABLED	The USB host port is disabled
PM_UNAVAILABLE	The status of the device could not be read.
PM_NO_CHANGE	The current value will be used

R

Macro	Definition
RA	Variable name: rate time
RE	Variable name: reset time
READ_COIL_STATUS	Modbus function code
READ_EXCEPTION_STATUS	Modbus function code
READ_HOLDING_REGISTER	Modbus function code
READ_INPUT_REGISTER	Modbus function code
READ_INPUT_STATUS	Modbus function code
READSTATUS	enum ReadStatus
REPORT_SLAVE_ID	Modbus function code
RS232	Specifies serial port is an RS-232 port.
RS232_COLLISION_AVOIDANCE	Specifies serial port is RS232 and uses CD for collision avoidance.
RS232_MODEM	Specifies serial port is an RS-232 dial-up modem.
RS485_4WIRE	Specifies serial port is a 4 wire RS-485 port.
RTOS_ENVELOPES	Number of RTOS envelopes.
RTOS_EVENTS	Number of RTOS events.
RTOS_PRIORITIES	Number of RTOS task priorities.
RTOS_RESOURCES	Number of RTOS resource flags.
RTOS_TASKS	Number of RTOS tasks.
RUN	Run/Service switch is in RUN position.

S

Macro	Definition
SP	Variable name: setpoint
SR	Variable name: status register
S_MODULE_FAILURE	Status LED code for I/O module communication failure
S_NORMAL	Status LED code for normal status
SCADAPACK	SCADAPack controller
SCADAPACK_LIGHT	SCADAPack LIGHT controller
SCADAPACK_PLUS	SCADAPack PLUS controller
SERIAL_PORTS	Number of serial ports.
SERVICE	Run/Service switch is in SERVICE position.

Macro	Definition
SF_ALREADY_DEFINED	Result code: translation is already defined in the table
SF_INDEX_OUT_OF_RANGE	Result code: invalid translation table index
SF_NO_TRANSLATION	Result code: entry does not define a translation
SF_PORT_OUT_OF_RANGE	Result code: serial port is not valid
SF_STATION_OUT_OF_RANGE	Result code: station number is not valid
SF_TABLE_SIZE	Number of entries in the store and forward table
SF_VALID	Result code: translation is valid
SIGNAL_CTS	I/O line bit mask: clear to send signal
SIGNAL_CTS	Matches status of CTS input.
SIGNAL_DCD	I/O line bit mask: carrier detect signal
SIGNAL_DCD	Matches status of DCD input.
SIGNAL_OFF	Specifies a signal is de-asserted
SIGNAL_OH	I/O line bit mask: off hook signal
SIGNAL_OH	Not supported – forced low (1).
SIGNAL_ON	Specifies a signal is asserted
SIGNAL_RING	I/O line bit mask: ring signal
SIGNAL_RING	Not supported – forced low (0).
SIGNAL_VOICE	I/O line bit mask: voice/data switch signal
SIGNAL_VOICE	Not supported – forced low (0).
SLEEP_MODE_SUPPORTED	Defined if sleep function is supported
SMARTWIRE_5201_5202	SmartWIRE 5201 and 5202 controllers
SP	Variable name: setpoint
SR	Variable name: status register
STACK_SIZE	Size of the machine stack.
START_COIL	Start of the coils section in the linear database.
START_HOLDING	Start of the holding register section in the linear database.
START_INPUT	Start of the input register section in the linear database.
START_STATUS	Start of the status section in the linear database.
STARTUP_ APPLICATION	Specifies the application start up task.
STARTUP_SYSTEM	Specifies the system start up task.
STOP1	Specifies 1 stop bit.

Macro	Definition
STOP2	Specifies 2 stop bits.
SYSTEM	Specifies a system type task. System tasks are not terminated by the end_application function.

T

Macro	Definition
T_CELSIUS	Specifies temperatures in degrees Celsius
T_FAHRENHEIT	Specifies temperatures in degrees Fahrenheit
T_KELVIN	Specifies temperatures in degrees Kelvin
T_RANKINE	Specifies temperatures in degrees Rankine
TELESAFE_6000_16EX	6000-16EX controller
TELESAFE_MICRO_16	Micro16 controller
TIMED_OUT	Specifies timer is has reached zero.
TIMEOUT	Specifies timer is generating timed output change.
TIMER_BADADDR	Error code: invalid digital I/O address
TIMER_BADINTERVAL	Error code: invalid timer interval
TIMER_BADTIMER	Error code: invalid timer
TIMER_BADVALUE	Error code: invalid time value
TIMER_MAX	Number of last valid software timer.
TS_EXECUTING	Task status indicating task is executing.
TS_READY	Task status indicating task is ready to execute
TS_WAIT_ RESOURCE	Task status indicating task is blocked waiting for a resource
TS_WAIT_ENVELOPE	Task status indicating task is blocked waiting for an envelope
TS_WAIT_EVENT	Task status indicating task is blocked waiting for an event
TS_WAIT_MESSAGE	Task status indicating task is blocked waiting for a message

V

Macro	Definition
VI_DATE_SIZE	Number of characters in version information date field

W

Macro	Definition
WRITESTATUS	enum WriteStatus
WS_ALL	All wake up sources enabled
WS_COUNTER_0_OVERFLOW	Bit mask to enable counter 0 overflow as wake up source
WS_COUNTER_1_OVERFLOW	Bit mask to enable counter 1 overflow as wake up source
WS_COUNTER_2_OVERFLOW	Bit mask to enable counter 2 overflow as wake up source
WS_INTERRUPT_INPUT	Bit mask to enable interrupt input as wake up source
WS_LED_POWER_SWITCH	Bit mask to enable LED power switch as wake up source
WS_NONE	No wake up source enabled
WS_REAL_TIME_CLOCK	Bit mask to enable real time clock as wake up source
WS_UNDEFINED	Undefined wake up source

Z

Macro	Definition
ZE	Variable name: zero scale output limit

Telepace C Tools Structures and Types

ABConfiguration

The ABConfiguration structure defines settings for DF1 communication protocol.

```
/* DF1 Protocol Configuration */
struct ABConfiguration {
       unsigned min_protected_address;
       unsigned max_protected_address;
       };
```

- min_protected_address is the minimum allowable DF1 physical 16-bit address allowed in all protected commands. The default value is 0.
- max_protected_address is the maximum allowable DF1 physical 16-bit address allowed in all protected commands. The default value is NUMAB.

ADDRESS_MODE

The ADDRESS_MODE enumerated type describes addressing modes for communication protocols.

```
typedef enum addressMode_t
       {
       AM_standard = 0,
       AM_extended
       }
       ADDRESS_MODE;
```

- AM_standard returns standard Modbus addressing. Standard addressing allows 255 stations and is compatible with standard Modbus devices
- AM_extended returns extended addressing. Extended addressing allows 65534 stations.

ALARM_SETTING

The ALARM_SETTING structure defines a real time clock alarm setting.

```
typedef struct alarmSetting_tag {
       UINT16 type;
       UINT16 hour;
       UINT16 minute;
       UINT16 second;
       } ALARM_SETTING;
```

- type specifies the type of alarm. It may be the AT_NONE or AT_ABSOLUTE macro.
- hour specifies the hour at which the alarm will occur.
- minute specifies the minute at which the alarm will occur.
- second specifies the second at which the alarm will occur.

clock

The clock structure contains time and date for reading or writing the real time clock.

```
struct clock {
    UINT16 year;
    UINT16 month;
    UINT16 day;
    UINT16 dayofweek;
    UINT16 hour;
    UINT16 minute;
    UINT16 second;
    };
```

- year is the current year. It is two digits in the range 00 to 99.
- month is the current month. It is in the range 1 to 12.
- day is the current day. It is in the range 1 to 31.
- dayofweek is the current day of the week. It is in the range 1 to 7. 1 = Sunday, 2 = Monday…7 = Saturday.
- hour is the current hour. It is in the range 00 to 23.
- minute is the current minute. It is in the range 00 to 59.
- second is the current second. It is in the range 00 to 59.

DATALOG_CONFIGURATION

The data log configuration structure holds the configuration of the data log. Each record in a data log may hold up to eight fields. Not all the fields are used if fewer than eight variables are declared.

The amount of memory used for a record depends on the number of fields in the record and the size of each field. Use the datalogRecordSize function to determine the memory needed for each record.

```
typedef struct datalogConfig_type {
      UINT16 records;     /* # of records */
      UINT16 fields;      /* # of fields per record */
      DATALOG_VARIABLE typesOfFields[MAX_NUMBER_OF_FIELDS];
} DATALOG_CONFIGURATION;
```

DATALOG_STATUS

The data log status enumerated type is used to report status information.

```
typedef enum {
       DLS_CREATED,  /* data log created */
       DLS_BADID,    /* invalid log ID */
       DLS_EXISTS,   /* log already exists */
       DLS_NOMEMORY, /* insufficient memory for log */
       DLS_BADCONFIG /* invalid configuration */
       DLS_BADSEQUENCE     /* sequence number not in use */
} DATALOG_STATUS;
```

DATALOG_VARIABLE

The data log variable enumerated type is specify the type and size of variables to be recorded in the log.

```
typedef enum {
       DLV_UINT16 = 0,     /* 16 bit unsigned integer */
       DLV_INT16,          /* 16 bit signed integer */
       DLV_UINT32,         /* 32 bit unsigned integer */
       DLV_INT32,          /* 32 bit signed integer */
       DLV_FLOAT,          /* 32 bit floating point */
       DLV_CMITIME,  /* 64 bit time */
       DLV_DOUBLE          /* 64 bit floating point */
} DATALOG_VARIABLE;
```

DialError

The DialError enumerated type defines error responses from the dial-up modem functions and may have one of the following values.

```
enum DialError
{
       DE_NoError = 0,
       DE_BadConfig,
       DE_NoModem,
       DE_InitError,
       DE_NoDialTone,
       DE_BusyLine,
       DE_CallAborted,
       DE_FailedToConnect,
       DE_CarrierLost,
       DE_NotInControl
       DE_CallCut
};
```

- DE_NoError returns no error has occurred
- DE_BadConfig returns the modem configuration structure contains an error
- DE_NoModem returns the serial port is not configured as a modem (port type must be RS232_MODEM). Or no modem is connected to the controller serial port.
- DE_InitError returns modem initialization failed (the modem may be turned off)

- DE_NoDialTone returns modem did not detect a dial tone or the S6 setting in the modem is too short.
- DE_BusyLine returns the phone number called was busy
- DE_CallAborted returns a call in progress was aborted by the user
- DE_FailedToConnect returns the modem could not connect to the remote site
- DE_CarrierLost returns the connection to the remote site was lost (modem reported NO CARRIER). Carrier is lost for a time exceeding the S10 setting in the modem. Phone lines with call waiting are very susceptible to this condition.
- DE_NotInControl returns the serial port is in use by another modem function or has answered an incoming call.
- DE_CallCut returns an incoming call was disconnected while attempting to dial out.

DialState

The DialState enumerated type defines the state of the modemDial operation and may have one of the following values.

enum DialState

```
{
       DS_Inactive,
       DS_Calling,
       DS_Connected,
       DS_Terminating
};
```

- DS_Inactive returns the serial port is not in use by a modem
- DS_Calling returns the controller is making a connection to a remote controller
- DS_Connected returns the controller is connected to a remote controller
- DS_Terminating returns the controller is ending a connection to a remote controller.

dnpAnalogInput

The dnpAnalogInput type describes a DNP analog input point. This type is used for both 16-bit and 32-bit points.

```
typedef struct dnpAnalogInput_type
{
       UINT16 modbusAddress;
       UCHAR  class;
       UINT32 deadband;
 } dnpAnalogInput;
```

- modbusAddress is the address of the Modbus register number associated with the point.

- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.
- deadband is the amount by which the analog input value must change before an event will be reported for the point.

dnpAnalogOutput

The dnpAnalogOutput type describes a DNP analog output point. This type is used for both 16-bit and 32-bit points.

```
typedef struct dnpAnalogOutput_type
{
       UINT16 modbusAddress;
} dnpAnalogOutput;
```

- modbusAddress is the address of the Modbus register associated with the point.

dnpBinaryInput

The dnpBinaryInput type describes a DNP binary input point.

```
typedef struct dnpBinaryInput_type
{
       UINT16 modbusAddress;
       UCHAR  class;
} dnpBinaryInput;
```

- modbusAddress is the address of the Modbus register associated with the point.
- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.

DNP Binary Input Extended Point

The dnpBinaryInputEx type describes an extended DNP Binary Input point.

```
typedef struct dnpBinaryInputEx_type
{
       UINT16 modbusAddress;
       UCHAR  eventClass;
       UCHAR  debounce;
} dnpBinaryInputEx;
```

- modbusAddress is the address of the Modbus register associated with the point.
- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.
- debounceTime is the debounce time for thebinary input.

dnpBinaryOutput

The dnpBinaryOutput type describes a DNP binary output point.

```
typedef struct dnpBinaryOutput_type
{
       UINT16 modbusAddress1;
       UINT16 modbusAddress2;
       UCHAR  controlType;
 } dnpBinaryOutput;
```

- modbusAddress1 is the address of the first Modbus register associated with the point. This field is always used.
- modbusAddress2 is the address of the second Modbus register associated with the point. This field is used only with paired outputs. See the controlType field.
- controlType determines if one or two outputs are associated with this output point. It may be set to PAIRED or NOT_PAIRED.
- A paired output uses two Modbus registers for output. The first output is the Trip output and the second is the Close output. This is used with Control Relay Output Block objects.
- A non-paired output uses one Modbus register for output. This is used with Binary Output objects.

DNP_CONNECTION_EVENT Type

This enumerated type lists DNP events.

```
typedef enum dnpConnectionEventType
{
       DNP_CONNECTED=0,
       DNP_DISCONNECTED,
       DNP_CONNECTION_REQUIRED,
       DNP_MESSAGE_COMPLETE,
       DNP_MESSAGE_TIMEOUT
} DNP_CONNECTION_EVENT;
```

- The DNP_CONNECTED event indicates that the handler has connected to the master station. The application sends this event to DNP. When DNP receives this event it will send unsolicited messages.
- The DNP_DISCONNECTED event indicates that the handler has disconnected from the master station. The application sends this event to DNP. When DNP receives this event it will request a new connection before sending unsolicited messages.
- The DNP_CONNECTION_REQUIRED event indicates that DNP wishes to connect to the master station. DNP sends this event to the application. The application should process this event by making a connection.
- The DNP_MESSAGE_COMPLETE event indicates that DNP has received confirmation of unsolicited messages from the master station. DNP sends this event to the application. The application should process this event by disconnecting. In many applications a short delay before disconnecting is

useful as it allows the master station to send commands to the slave after the unsolicited reporting is complete.

- The DNP_MESSAGE_TIMEOUT event indicates that DNP has attempted to send an unsolicited message but did not receive confirmation after all attempts. This usually means there is a communication problem. DNP sends this event to the application. The application should process this event by disconnecting.

dnpConfiguration

The dnpConfiguration type describes the DNP parameters.

```
typedef struct dnpConfiguration_type
{
       UINT16 masterAddress;
       UINT16 rtuAddress;
       CHAR   datalinkConfirm;
       CHAR   datalinkRetries;
       UINT16 datalinkTimeout;
       UINT16 operateTimeout;
       UCHAR  applicationConfirm;
       UINT16 maximumResponse;
       UCHAR  applicationRetries;
       UINT16 applicationTimeout;
       INT16  timeSynchronization;
       UINT16 BI_number;
UINT16 BI_startAddress;
       CHAR   BI_reportingMethod;
       UINT16 BI_soebufferSize;
       UINT16 BO_number;
UINT16 BO_startAddress;
       UINT16 CI16_number;
UINT16 CI16_startAddress;
       CHAR   CI16_reportingMethod;
       UINT16 CI16_bufferSize;
       UINT16 CI32_number;
UINT16 CI32_startAddress;
       CHAR   CI32_reportingMethod;
       UINT16 CI32_bufferSize;
CHAR   CI32_wordOrder;
       UINT16 AI16_number;
UINT16 AI16_startAddress;
       CHAR   AI16_reportingMethod;
       UINT16 AI16_bufferSize;
       UINT16 AI32_number;
UINT16 AI32_startAddress;
       CHAR   AI32_reportingMethod;
       UINT16 AI32_bufferSize;
CHAR   AI32_wordOrder;
       UINT16 AISF_number;
UINT16 AISF_startAddress;
       CHAR   AISF_reportingMethod;
       UINT16 AISF_bufferSize;
CHAR   AISF_wordOrder;
       UINT16 AO16_number;
```

```
UINT16 AO16_startAddress;
       UINT16 AO32_number;
UINT16 AO32_startAddress;
CHAR   AO32_wordOrder;
       UINT16 AOSF_number;
UINT16 AOSF_startAddress;
CHAR   AOSF_wordOrder;
       UINT16 autoUnsolicitedClass1;
       UINT16 holdTimeClass1;
       UINT16 holdCountClass1;
       UINT16 autoUnsolicitedClass2;
       UINT16 holdTimeClass2;
       UINT16 holdCountClass2;
       UINT16 autoUnsolicitedClass3;
       UINT16 holdTimeClass3;
       UINT16 holdCountClass3;
} dnpConfiguration;
```

- masterAddress is the address of the master station. Unsolicited messages are sent to this station. Solicited messages must come from this station. Valid values are 0 to 65534.
- rtuAddress is the address of the RTU. The master station must send messages to this address. Valid values are 0 to 65534.
- datalinkConfirm enables requesting data link layer confirmations. Valid values are TRUE and FALSE.
- datalinkRetries is the number of times the data link layer will retry a failed message. Valid values are 0 to 255.
- datalinkTimeout is the length of time the data link layer will wait for a response before trying again or aborting the transmission. The value is measured in milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds.
- operateTimeout is the length of time an operate command is valid after receiving a select command. The value is measured in seconds. Valid values are 1 to 6500.
- applicationConfirm enables requesting application layer confirmations. Valid values are TRUE and FALSE.
- maximumResponse is the maximum length of an application layer response. Valid values are 20 to 2048. The recommended value is 2048 unless the master cannot handle responses this large.
- applicationRetries is the number of times the application layer will retry a transmission. Valid values are 0 to 255.
- applicationTimeout is the length of time the application layer will wait for a response before trying again or aborting the transmission. The value is measured in milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds. This value must be larger than the data link timeout.

- timeSynchronization defines how often the RTU will request a time synchronization from the master.
- Set this to NO_TIME_SYNC to disable time synchronization requests.
- Set this to STARTUP_TIME_SYNC to request time synchronization at start up only.
- Set this to 1 to 32767 to set the time synchronization period in seconds.
- BI_number is the number of binary input points. Valid values are 0 to 9999.
- BI_startAddress is the DNP address of the first Binary Input point.
- BI_reportingMethod determines how binary inputs are reported either Change Of State or Log All Events.
- BI_bufferSize is the Binary Input Change Event Buffer Size.
- BO_number is the number of binary output points. Valid values are 0 to 9999.
- BO_startAddress is the DNP address of the first Binary Output point.
- CI16_number is the number of 16-bit counter input points. Valid values are 0 to 9999.
- CI16_startAddress is the DNP address of the first CI16 point.
- CI16_reportingMethod determines how CI16 inputs are reported either Change Of State or Log All Events.
- CI16_bufferSize is the number of events in the 16-bit counter change buffer. Valid values are 0 to 9999.
- CI32_number is the number of 32-bit counter input points. Valid values are 0 to 9999.
- CI32_startAddress is the DNP address of the first CI32 point.
- CI32_reportingMethod determines how CI32 inputs are reported either Change Of State or Log All Events.
- CI32_bufferSize is the number of events in the 32-bit counter change buffer. Valid values are 0 to 9999.
- CI32_wordOrder is the Word Order of CI32 points (0=LSW first, 1=MSW first).
- AI16_number is the number of 16-bit analog input points. Valid values are 0 to 9999.
- AI16_startAddress is the DNP address of the first AI16 point.
- AI16_reportingMethod determines how 16-bit analog changes are reported.
- Set this to FIRST_VALUE to report the value of the first change event measured.

- Set this to CURRENT_VALUE to report the value of the latest change event measured.
- AI16_bufferSize is the number of events in the 16-bit analog input change buffer. Valid values are 0 to 9999.
- AI32_number is the number of 32-bit analog input points. Valid values are 0 to 9999.
- AI32_startAddress is the DNP address of the first AI32 point.
- AI32_reportingMethod determines how 32-bit analog changes are reported.
- Set this to FIRST_VALUE to report the value of the first change event measured.
- Set this to CURRENT_VALUE to report the value of the latest change event measured.
- AI32_bufferSize is the number of events in the 32-bit analog input change buffer. Valid values are 0 to 9999.
- AI32_wordOrder is the Word Order of AI32 points (0=LSW first, 1=MSW first)
- AO16_number is the number of 16-bit analog output points. Valid values are 0 to 9999.
- AO16_startAddress is the DNP address of the first AO16 point.
- AO32_number is the number of 32-bit analog output points. Valid values are 0 to 9999.
- AO32_startAddress is the DNP address of the first AO32 point.
- AO32_wordOrder is the Word Order of AO32 points (0=LSW first, 1=MSW first)
- AOSF_number is the number of short float Analog Outputs.
- AOSF_startAddress is the DNP address of first AOSF point.
- AOSF_wordOrder is the Word Order of AOSF points (0=LSW first, 1=MSW first).
- autoUnsolicitedClass1 enables or disables automatic Unsolicited reporting of Class 1 events.
- holdTimeClass1 is the maximum period to hold Class 1 events before reporting
- holdCountClass1 is the maximum number of Class 1 events to hold before reporting.
- autoUnsolicitedClass2 enables or disables automatic Unsolicited reporting of Class 2 events.
- holdTimeClass2 is the maximum period to hold Class 2 events before reporting

- holdCountClass2 is the maximum number of Class 2 events to hold before reporting.
- autoUnsolicitedClass3 enables or disables automatic Unsolicited reporting of Class 3 events.
- holdTimeClass3 is the maximum period to hold Class 3 events before reporting.
- holdCountClass2 is the maximum number of Class 3 events to hold before reporting.

dnpConfigurationEx

The dnpConfigurationEx type includes extra parameters in the DNP Configuration.

```
typedef struct dnpConfigurationEx_type
{
       UINT16 rtuAddress;
       UCHAR  datalinkConfirm;
       UCHAR  datalinkRetries;
       UINT16 datalinkTimeout;
       UINT16 operateTimeout;
       UCHAR  applicationConfirm;
       UINT16 maximumResponse;
       UCHAR  applicationRetries;
       UINT16 applicationTimeout;
       INT16  timeSynchronization;
       UINT16 BI_number;
       UINT16 BI_startAddress;
       UCHAR  BI_reportingMethod;
       UINT16 BI_soeBufferSize;
       UINT16 BO_number;
       UINT16 BO_startAddress;
       UINT16 CI16_number;
       UINT16 CI16_startAddress;
       UCHAR  CI16_reportingMethod;
       UINT16 CI16_bufferSize;
       UINT16 CI32_number;
       UINT16 CI32_startAddress;
       UCHAR  CI32_reportingMethod;
       UINT16 CI32_bufferSize;
       UCHAR  CI32_wordOrder;
       UINT16 AI16_number;
       UINT16 AI16_startAddress;
       UCHAR  AI16_reportingMethod;
       UINT16 AI16_bufferSize;
       UINT16 AI32_number;
       UINT16 AI32_startAddress;
       UCHAR  AI32_reportingMethod;
       UINT16 AI32_bufferSize;
       UCHAR  AI32_wordOrder;
       UINT16 AISF_number;
       UINT16 AISF_startAddress;
       UCHAR  AISF_reportingMethod;
```

```
        UINT16 AISF_bufferSize;
        UCHAR  AISF_wordOrder;
        UINT16 AO16_number;
        UINT16 AO16_startAddress;
        UINT16 AO32_number;
        UINT16 AO32_startAddress;
        UCHAR  AO32_wordOrder;
        UINT16 AOSF_number;
        UINT16 AOSF_startAddress;
        UCHAR  AOSF_wordOrder;
        UINT16 autoUnsolicitedClass1;
        UINT16 holdTimeClass1;
        UINT16 holdCountClass1;
        UINT16 autoUnsolicitedClass2;
        UINT16 holdTimeClass2;
        UINT16 holdCountClass2;
        UINT16 autoUnsolicitedClass3;
        UINT16 holdTimeClass3;
        UINT16 holdCountClass3;
        UINT16 enableUnsolicitedOnStartup;
        UINT16 sendUnsolicitedOnStartup;
        UINT16 level2Compliance;
        UINT16 masterAddressCount;
        UINT16 masterAddress[8];
        UINT16 maxEventsInResponse;
        UINT16 dialAttempts;
        UINT16 dialTimeout;
        UINT16 pauseTime;
        UINT16 onlineInactivity;
        UINT16 dialType;
        Char   modemInitString[64];
} dnpConfigurationEx;
```

- rtuAddress is the address of the RTU. The master station must send messages to this address. Valid values are 0 to 65534.
- datalinkConfirm enables requesting data link layer confirmations. Valid values are TRUE and FALSE.
- datalinkRetries is the number of times the data link layer will retry a failed message. Valid values are 0 to 255.
- datalinkTimeout is the length of time the data link layer will wait for a response before trying again or aborting the transmission. The value is measured in milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds.
- operateTimeout is the length of time an operate command is valid after receiving a select command. The value is measured in seconds. Valid values are 1 to 6500.
- applicationConfirm enables requesting application layer confirmations. Valid values are TRUE and FALSE.
- maximumResponse is the maximum length of an application layer response. Valid values are 20 to 2048. The recommended value is 2048 unless the master cannot handle responses this large.

- applicationRetries is the number of times the application layer will retry a transmission. Valid values are 0 to 255.
- applicationTimeout is the length of time the application layer will wait for a response before trying again or aborting the transmission. The value is measured in milliseconds. Valid values are 100 to 60000 in multiples of 100 milliseconds. This value must be larger than the data link timeout.
- timeSynchronization defines how often the RTU will request a time synchronization from the master.
- Set this to NO_TIME_SYNC to disable time synchronization requests.
- Set this to STARTUP_TIME_SYNC to request time synchronization at start up only.
- Set this to 1 to 32767 to set the time synchronization period in seconds.
- BI_number is the number of binary input points. Valid values are 0 to 9999.
- BI_startAddress is the DNP address of the first Binary Input point.
- BI_reportingMethod determines how binary inputs are reported either Change Of State or Log All Events.
- BI_soebufferSize is the Binary Input Change Event Buffer Size.
- BO_number is the number of binary output points. Valid values are 0 to 9999.
- BO_startAddress is the DNP address of the first Binary Output point.
- CI16_number is the number of 16-bit counter input points. Valid values are 0 to 9999.
- CI16_startAddress is the DNP address of the first CI16 point.
- CI16_reportingMethod determines how CI16 inputs are reported either Change Of State or Log All Events.
- CI16_bufferSize is the number of events in the 16-bit counter change buffer. Valid values are 0 to 9999.
- CI32_number is the number of 32-bit counter input points. Valid values are 0 to 9999.
- CI32_startAddress is the DNP address of the first CI32 point.
- CI32_reportingMethod determines how CI32 inputs are reported either Change Of State or Log All Events.
- CI32_bufferSize is the number of events in the 32-bit counter change buffer. Valid values are 0 to 9999.
- CI32_wordOrder is the Word Order of CI32 points (0=LSW first, 1=MSW first).

- AI16_number is the number of 16-bit analog input points. Valid values are 0 to 9999.
- AI16_startAddress is the DNP address of the first AI16 point.
- AI16_reportingMethod determines how 16-bit analog changes are reported.
- Set this to FIRST_VALUE to report the value of the first change event measured.
- Set this to CURRENT_VALUE to report the value of the latest change event measured.
- AI16_bufferSize is the number of events in the 16-bit analog input change buffer. Valid values are 0 to 9999.
- AI32_number is the number of 32-bit analog input points. Valid values are 0 to 9999.
- AI32_startAddress is the DNP address of the first AI32 point.
- AI32_reportingMethod determines how 32-bit analog changes are reported.
- Set this to FIRST_VALUE to report the value of the first change event measured.
- Set this to CURRENT_VALUE to report the value of the latest change event measured.
- AI32_bufferSize is the number of events in the 32-bit analog input change buffer. Valid values are 0 to 9999.
- AI32_wordOrder is the Word Order of AI32 points (0=LSW first, 1=MSW first)
- AISF_number is the number of short float Analog Inputs.
- AISF_startAddress is the DNP address of first AISF point.
- AISF_reportingMethod is the event reporting method, Change Of State or Log All Events.
- AISF_bufferSize is the short float Analog Input Event Buffer Size.
- AISF_wordOrder is the word order of AISF points (0=LSW first, 1=MSW first) */
- AO16_number is the number of 16-bit analog output points. Valid values are 0 to 9999.
- AO16_startAddress is the DNP address of the first AO16 point.
- AO32_number is the number of 32-bit analog output points. Valid values are 0 to 9999.
- AO32_startAddress is the DNP address of the first AO32 point.
- AO32_wordOrder is the Word Order of AO32 points (0=LSW first, 1=MSW first)

- AOSF_number is the number of short float Analog Outputs.
- AOSF_startAddress is the DNP address of first AOSF point.
- AOSF_wordOrder is the Word Order of AOSF points (0=LSW first, 1=MSW first).
- autoUnsolicitedClass1 enables or disables automatic Unsolicited reporting of Class 1 events.
- holdTimeClass1 is the maximum period to hold Class 1 events before reporting
- holdCountClass1 is the maximum number of Class 1 events to hold before reporting.
- autoUnsolicitedClass2 enables or disables automatic Unsolicited reporting of Class 2 events.
- holdTimeClass2 is the maximum period to hold Class 2 events before reporting
- holdCountClass2 is the maximum number of Class 2 events to hold before reporting.
- autoUnsolicitedClass3 enables or disables automatic Unsolicited reporting of Class 3 events.
- holdTimeClass3 is the maximum period to hold Class 3 events before reporting.
- HoldCountClass3 is the maximum number of Class 3 events to hold before reporting.
- EnableUnsolicitedOnStartup enables or disables unsolicited reporting at start-up.
- SendUnsolicitedOnStartup sends an unsolicited report at start-up.
- level2Compliance reports only level 2 compliant data types (excludes floats, AO-32).
- MasterAddressCount is the number of master stations.
- masterAddress[8] is the number of master station addresses.
- MaxEventsInResponse is the maximum number of change events to include in read response.
- PSTNDialAttempts is the maximum number of dial attempts to establish a PSTN connection.
- PSTNDialTimeout is the maximum time after initiating a PSTN dial sequence to wait for a carrier signal.
- PSTNPauseTime is the pause time between dial events.

- PSTNOnlineInactivity is the maximum time after message activity to leave a PSTN connection open before hanging up.
- PSTNDialType is the dial type: tone or pulse dialling.
- modemInitString[64] is the initialization string to send to the modem.

dnpCounterInput

The dnpCounterInput type describes a DNP counter input point. This type is used for both 16-bit and 32-bit points.

```
typedef struct dnpCounterInput_type
{
       UINT16 modbusAddress;
       UCHAR  class;
       UINT32 threshold;
 } dnpCounterInput;
```

- modbusAddress is the address of the Modbus register number associated with the point.
- class is the reporting class for the object. It may be set to CLASS_1, CLASS_2 or CLASS_3.
- threshold is the amount by which the counter input value must change before an event will be reported for the point.

dnpPointType

The enumerated type DNP_POINT_TYPE includes all allowed DNP data point types.

```
typedef enum dnpPointType
{
       BI_POINT=0,          /* binary input */
       AI16_POINT,          /* 16 bit analog input */
       AI32_POINT,          /* 32 bit analog input */
       AISF_POINT,          /* short float analog input */
       AILF_POINT,          /* long float analog input */
       CI16_POINT,          /* 16 bit counter output */
       CI32_POINT,          /* 32 bit counter output */
       BO_POINT,            /* binary output */
       AO16_POINT,          /* 16 bit analog output */
       AO32_POINT,          /* 32 bit analog output */
       AOSF_POINT,          /* short float analog output */
       AOLF_POINT           /* long float analog output */
} DNP_POINT_TYPE;
```

DNP_RUNTIME_STATUS

The DNP_RUNTIME_STATUS type describes a structure for holding status information about DNP event log buffers.

```
/* DNP Runtime Status */
typedef struct dnp_runtime_status
{
       UINT16 eventCountBI;
```

```
        UINT16 eventCountCI16;
        UINT16 eventCountCI32;
        UINT16 eventCountAI16;
        UINT16 eventCountAI32;
        UINT16 eventCountAISF;
        UINT16 eventCountClass1;
        UINT16 eventCountClass2;
        UINT16 eventCountClass3;
} DNP_RUNTIME_STATUS;
```

- eventCountBI is number of binary input events.
- eventCountCI16 is number of 16-bit counter events.
- eventCountCI32 is number of 32-bit counter events.
- eventCountAI16 is number of 16-bit analog input events.
- eventCountAI32 is number of 32-bit analog input events.
- eventCountAISF is number of short floating-point analog input events.
- eventCountClass1 is the class 1 event counter.
- eventCountClass2 is the class 2 event counter.
- eventCountClass3 is the class 3 event counter.

envelope

The envelope type is a structure containing a message envelope. Envelopes are used for inter-task communication.

```
typedef struct env {
    struct env    *link;
    unsigned      source;
    unsigned      destination;
    unsigned      type;
    unsigned long data;
    unsigned      owner;
    }
    envelope;
```

- link is a pointer to the next envelope in a queue. This field is used by the RTOS. It is of no interest to an application program.
- source is the task ID of the task sending the message. This field is specified automatically by the send_message function. The receiving task may read this field to determine the source of the message.
- destination is the task ID of the task to receive the message. It must be specified before calling the send_message function.
- type specifies the type of data in the data field. It may be MSG_DATA, MSG_POINTER, or any other value defined by the application program. This field is not required.

- data is the message data. The field may contain a datum or pointer. The application program determines the use of this field.
- owner is the task that owns the envelope. This field is set by the RTOS and must not be changed by an application program.

HART_COMMAND

The HART_COMMAND type is a structure containing a command to be sent to a HART slave device. The command field contains the HART command number. The length field contains the length of the data string to be transmitted (the byte count in HART documentation). The data field contains the data to be sent to the slave.

```
typedef struct hartCommand_t
       {
       unsigned command;
       unsigned length;
       char     data[DATA_SIZE];
       }
       HART_COMMAND;
```

- command is the HART command number.
- length is the number of characters in the data string.
- data[DATA_SIZE] is the data field for the command.

HART_DEVICE

The HART_DEVICE type is a structure containing information about the HART device. The information is read from the device using command 0 or command 11. The fields are identical to those read by the commands. Refer to the command documentation for more information.

```
typedef struct hartDevice_t
       {
       unsigned char manufacturerID;
       unsigned char manufacturerDeviceType;
       unsigned char preamblesRequested;
       unsigned char commandRevision;
       unsigned char transmitterRevision;
       unsigned char softwareRevision;
       unsigned char hardwareRevision;
       unsigned char flags;
       unsigned long deviceID;
       }
       HART_DEVICE;
```

HART_RESPONSE

The HART_RESPONSE type is a structure containing a response from a HART slave device. The command field contains the HART command number. The length field contains the length of the data string to be transmitted (the byte count in HART documentation). The data field contains the data to be sent to the slave.

```
typedef struct hartResponse_t
      {
      unsigned responseCode,
      unsigned length,
      char     data[DATA_SIZE];
      }
      HART_RESPONSE;
```

- response is the response code from the device.
- length is the length of response data.
- data[DATA_SIZE] is the data field for the response.

HART_RESULT

The HART_RESULT enumeration type defines a list of results of sending a command.

```
typedef enum hartResult_t
      {
      HR_NoModuleResponse=0,
      HR_CommandPending,
      HR_CommandSent,
      HR_Response,
      HR_NoResponse,
      HR_WaitTransmit
      }
      HART_RESULT;
```

- HR_NoModuleResponse returns no response from HART modem module.
- HR_CommandPending returns command ready to be sent, but not sent.
- HR_CommandSent returns command sent.
- HR_Response returns response received.
- HR_NoResponse returns no response after all attempts.
- HR_WaitTransmit returns modem is not ready to transmit.

HART_SETTINGS

The HART_SETTINGS type is a structure containing the configuration for the HART modem module. The useAutoPreamble field indicates if the number of preambles is set by the value in the HART_SETTINGS structure (FALSE) or the value in the HART_DEVICE structure (TRUE). The deviceType field determines if the 5904 modem is a HART primary master or secondary master device (primary master is the recommended setting).

```
typedef struct hartSettings_t
      {
      unsigned attempts;
      unsigned preambles;
      BOOLEAN  useAutoPreamble;
      unsigned deviceType;
```

```
        }
        HART_SETTINGS;
```

- attempts is the number of command attempts (1 to 4).
- preambles is the number of preambles to send (2 to 15).
- useAutoPreamble is a flag to use the requested preambles.
- deviceType is the type of HART master (1 = primary; 0 = secondary).

HART_VARIABLE

The HART_VARIABLE type is a structure containing a variable read from a HART device. The structure contains three fields that are used by various commands. Note that not all fields will be used by all commands. Refer to the command specific documentation.

```
typedef struct hartVariable_t
        {
        float    value;
        unsigned units;
        unsigned variableCode;
        }
        HART_VARIABLE;
```

- value is the value of the variable.
- units are the units of measurement.
- variableCode is the transmitter specific variable ID.

ioModules

The ioModules enumerated type describes I/O modules used with register assignment.

```
enum ioModules
{
        DOUT_generic8 = 0,
        DOUT_generic16,
        DOUT_5401,
        DOUT_5402,
        DOUT_5406,
        DOUT_5407,
        DOUT_5408,
        DOUT_5409,
        DOUT_5411,
        CNFG_clearPortCounters,
        CNFG_clearProtocolCounters,
        CNFG_saveToEEPROM,
        CNFG_LEDPower,

        SCADAPack_lowerIO,
        SCADAPack_upperIO,

        DIN_generic8,
        DIN_generic16,
        DIN_5401,
```

```
        DIN_5402,
        DIN_5403,
        DIN_5404,
        DIN_5405,
        DIN_5421,
        DIN_520xDigitalInputs,
        DIN_520xOptionSwitches,
        DIN_520xInterruptInput,
        DIAG_forceLED,

        AIN_generic8,
        AIN_5501,
        AIN_5503,
        AIN_5504,
        AIN_5521,
        CNTR_5410,
        CNTR_520xCounterInputs,
        AIN_520xTemperature,
        AIN_520xRAMBattery,
        DIAG_controllerStatus,
        DIAG_commStatus,
        DIAG_protocolStatus,

        AOUT_generic2,
        AOUT_generic4,
        AOUT_5301,
        AOUT_5302,
        SCADAPack_AOUT,
        CNFG_portSettings,
        CNFG_protocolSettings,
        CNFG_realTimeClock,
        CNFG_PIDBlock,
        CNFG_storeAndForward,
        CNFG_5904Modem,
        CNFG_protocolExtended,
        AIN_5502,
        CNTR_520xInterruptInput,
CNFG_setSerialPortDTR,
        SCADAPack_LPIO,
        SCADAPack_10
CNFG_protocolExtendedEx,
        SCADAPack_5604IO,
        AOUT_5304,
GFC_4202IO
};
```

ledControl_tag

The ledControl_tag structure defines LED power control parameters.

```
struct ledControl_tag {
        unsigned state;
        unsigned time;
        };
```

- state is the default LED state. It is either the LED_ON or LED_OFF macro.

- time is the period, in minutes, after which the LED power returns to its default state.

ModemInit

The ModemInit structure specifies modem initialization parameters for the modemInit function.

```
struct ModemInit
{
       FILE * port;
       char   modemCommand[MODEM_CMD_MAX_LEN + 2];
};
```

- port is the serial port where the modem is connected.
- modemCommand is the initialization string for the modem. The characters AT will be prefixed to the command, and a carriage returned suffixed to the command when it is sent to the modem. Refer to the section **Modem Commands** for suggested command strings for your modem.

ModemSetup

The ModemSetup structure specifies modem initialization and dialing control parameters for the modemDial function.

```
struct ModemSetup
{
       FILE * port;
       unsigned short      dialAttempts;
       unsigned short      detectTime;
       unsigned short      pauseTime;
       unsigned short      dialmethod;
       char modemCommand[MODEM_CMD_MAX_LEN + 2];
       char phoneNumber[PHONE_NUM_MAX_LEN + 2];
};
```

- port is the serial port where the modem is connected.
- dialAttempts is the number of times the controller will attempt to dial the remote controller before giving up and reporting an error.
- detectTime is the length of time in seconds that the controller will wait for carrier to be detected. It is measured from the start of the dialing attempt.
- pauseTime is the length of time in seconds that the controller will wait between dialing attempts.
- dialmethod selects pulse or tone dialing. Set dialmethod to 0 for tone dialing or 1 for pulse dialing.
- modemCommand is the initialization string for the modem. The characters AT will be prepended to the command, and a carriage returned appended to

the command when it is sent to the modem. Refer to the section **Modem Commands** for suggested command strings for your modem.

- phoneNumber is the phone number of the remote controller. The characters ATD and the dialing method will be prepended to the command, and a carriage returned appended to the command when it is sent to the modem.

PID_DATA

The PID_DATA structure defines settings for the PID function.

```
typedef struct pidData_type
      {
      float pv;
      float sp;
      float gain;
      float reset;
      float rate;
      float deadband;
      float fullScale;
      float zeroScale;
      float manualOutput;
      UINT32 period;
      BOOLEAN autoMode;

      /* calculation results */
      float output;
      BOOLEAN outOfDeadband;

      /* historic data values */
      float pvN1;
      float pvN2;
      float errorN1;
      UINT32 lastTime;
      }
      PID_DATA;
```

- pv is the process value.
- sp is the PID setpoint.
- gain is the PID gain value.
- reset is the PID reset time in seconds.
- rate is the PID rate time in seconds.
- deadband is the PID deadband.
- fullScale is the PID full scale output limit.
- zeroScale is the PID zero scale output limit.
- manualOutput is the PID manual output value.

- period is the PID execution period in milliseconds.
- autoMode is the PID automode flage; TRUE = auto mode.
- output is the PID last output value.
- outOfDeadband is the PID outside of deadband error.
- pvN1 is the process value from n-1 iteration.
- pvN2 is the process value from n-2 iteration.
- errorN1 is the error from n-1 iteration.
- lastTime is the time of the last PID execution.

PROTOCOL_SETTINGS

The Extended Protocol Settings structure defines settings for a communication protocol. This structure differs from the standard settings in that it allows additional settings to be specified.

```
typedef struct protocolSettings_t
       {
       unsigned char type;
       unsigned station;
       unsigned char priority;
       unsigned SFMessaging;
       ADDRESS_MODE mode;
       }
       PROTOCOL_SETTINGS;
```

- type is the protocol type. It may be one of NO_PROTOCOL, MODBUS_RTU, or MODBUS_ASCII macros.
- station is the station address of the controller. Note that each serial port may have a different address. The valid values are determined by the communication protocol. This field is not used if the protocol type is NO_PROTOCOL.
- priority is the task priority of the protocol task. This field is not used if the protocol type is NO_PROTOCOL.
- SFMessaging is the enable Store and Forward messaging control flag.
- ADDRESS_MODE is the addressing mode, standard or extended.

PROTOCOL_SETTINGS_EX Type

This structure contains serial port protocol settings including Enron Modbus support.

```
typedef struct protocolSettingsEx_t
       {
       UCHAR type;
       UINT16 station;
       UCHAR priority;
       UINT16 SFMessaging;
```

```
ADDRESS_MODE mode;
BOOLEAN enronEnabled;
UINT16 enronStation;
}
PROTOCOL_SETTINGS_EX;
```

- type is the protocol type. It may be one of NO_PROTOCOL, MODBUS_RTU, or MODBUS_ASCII.
- station is the station address of the controller. Note that each serial port may have a different address. The valid values are determined by the communication protocol. This field is not used if the protocol type is NO_PROTOCOL.
- priority is the task priority of the protocol task. This field is not used if the protocol type is NO_PROTOCOL.
- SFMessaging is the enable Store and Forward messaging control flag.
- ADDRESS_MODE is the addressing mode, AM_standard or AM_extended.
- enronEnabled determines if the Enron Modbus station is enabled. It may be TRUE or FALSE.
- enronStation is the station address for the Enron Modbus protocol. It is used if enronEnabled is set to TRUE. Valid values are 1 to 255 for standard addressing, and 1 to 65534 for extended addressing.

prot_settings

The Protocol Settings structure defines settings for a communication protocol. This structure differs from the extended settings in that it allows fewer settings to be specified.

```
struct prot_settings {
    unsigned char type;
    unsigned char station;
    unsigned char priority;
    unsigned SFMessaging;
    };
```

- type is the protocol type. It may be one of NO_PROTOCOL, MODBUS_RTU, MODBUS_ASCII, AB_FULL_BCC, AB_HALF_BCC, AB_FULL_CRC, AB_HALF_CRC or DNP macros.
- station is the station address of the controller. Note that each serial port may have a different address. The valid values are determined by the communication protocol. This field is not used if the protocol type is NO_PROTOCOL.
- priority is the task priority of the protocol task. This field is not used if the protocol type is NO_PROTOCOL.
- SFMessaging is the enable Store and Forward messaging control flag.

prot_status

The prot_status structure contains protocol status information.

```
struct prot_status {
    unsigned command_errors;
    unsigned format_errors;
    unsigned checksum_errors;
    unsigned cmd_received;
    unsigned cmd_sent;
    unsigned rsp_received;
    unsigned rsp_sent;
    unsigned command;
    int      task_id;
    unsigned stored_messages;
    unsigned forwarded_messages;
    };
```

- command_errors is the number of messages received with invalid command codes.
- format_errors is the number of messages received with bad message data.
- checksum_errors is the number of messages received with bad checksums.
- cmd_received is the number of commands received.
- cmd_sent is the number of commands sent by the master_message function.
- rsp_received is the number of responses received by the master_message function.
- rsp_sent is the number of responses sent.
- command is the status of the last protocol command sent.
- task_id is the ID of the protocol task. This field is used by the set_protocol function to control protocol execution.
- stored_messages is the number of messages stored for forwarding.
- forwarded_messages is the number of messages forwarded.

pconfig

The pconfig structure contains serial port settings.

```
struct pconfig {
    unsigned baud;
    unsigned duplex;
    unsigned parity;
    unsigned data_bits;
    unsigned stop_bits;
    unsigned flow_rx;
    unsigned flow_tx;
    unsigned type;
    unsigned timeout;
    };
```

- baud is the communication speed. It is one of the BAUD_xxx macros.
- duplex is either the FULL or HALF macro.
- parity is one of NONE, EVEN or ODD macros.
- data_bits is the word length. It is either the DATA7 or DATA8 macro.
- stop_bits in the number of stop bits transmitted. It is either the STOP1 or STOP2 macro.
- flow_rx specifies flow control on the receiver. It is either the DISABLE or ENABLE macro.
- For com1 and com2 setting this parameter selects XON/XOFF flow control. It may be enabled or disabled.

If any protocol, other than Modbus ASCII, is used on the port you must set flow_rx to DISABLE. If Modbus ASCII or no protocol is used, you can set flow_rx to ENABLE or DISABLE. In most cases DISABLE is recommended.

- For com3 and com4 setting this parameter selects Receiver Disable after message reception. This is used with the Modbus RTU protocol only. If the Modbus RTU protocol is used, set flow_rx to ENABLE. Otherwise set flow_rx to DISABLE.
- flow_tx specifies flow control on the transmitter. It is either the DISABLE or ENABLE macro.
- For com1 and com2 setting this parameter selects XON/XOFF flow control. It may be enabled or disabled.

If any protocol, other than Modbus ASCII, is used on the port you must set flow_tx to DISABLE. If Modbus ASCII or no protocol is used, you can set flow_tx to ENABLE or DISABLE. In most cases DISABLE is recommended.

- For com3 and com4 setting this parameter indicates if the port should ignore the CTS signal. Setting the parameter to ENABLE causes the port to ignore the CTS signal.
- type specifies the serial port type. It is one of NOTYPE, RS232, RS232_MODEM, RS485, or RS232_COLLISION_AVOID macros.
- timeout specifies the time the driver will wait when the transmit buffer fills, before it clears the buffer.

PORT_CHARACTERISTICS

The PORT_CHARACTERISTICS type is a structure that contains serial port characteristics.

```
typedef struct portCharacteristics_tag {
	unsigned dataflow;
	unsigned buffering;
	unsigned protocol;
	unsigned long options;
	} PORT_CHARACTERISTICS;
```

- dataflow is a bit mapped field describing the data flow options supported on the serial port. ANDing can isolate the options with the PC_FLOW_RX_RECEIVE_STOP, PC_FLOW_RX_XON_XOFF, PC_FLOW_TX_IGNORE_CTS or PC_FLOW_TX_XON_XOFF macros.
- buffering describes the buffering options supported. No buffering options are currently supported.
- protocol describes the protocol options supported. The macro, PC_PROTOCOL_RTU_FRAMING is the only option supported.
- options describes additional options supported. No additional options are currently supported.

pstatus

The pstatus structure contains serial port status information.

```
struct pstatus {
    unsigned framing;
    unsigned parity;
    unsigned c_overrun;
    unsigned b_overrun;
    unsigned rx_buffer_size;
    unsigned rx_buffer_used;
    unsigned tx_buffer_size;
    unsigned tx_buffer_used;
    unsigned io_lines;
    };
```

- framing is the number of received characters with framing errors.
- parity is the number of received characters with parity errors.
- c_overrun is the number of received character overrun errors.
- b_overrun is the number of receive buffer overrun errors.
- rx_buffer_size is the size of the receive buffer in characters.
- rx_buffer_used is the number of characters in the receive buffer.
- tx_buffer_size is the size of the transmit buffer in characters.
- tx_buffer_used is the number of characters in the transmit buffer.
- io_lines is a bit mapped field indicating the status of the I/O lines on the serial port. The values for these lines differ between serial ports (see tables below). ANDing can isolate the signals with the SIGNAL_CTS, SIGNAL_DCD, SIGNAL_OH, SIGNAL_RING or SIGNAL_VOICE macros.

READSTATUS

The READSTATUS enumerated type indicates the status of an I^2C bus message read and may have one of the following values.

```
enum ReadStatus {
	RS_success,
	RS_selectFailed
	};
```
typedef enum ReadStatus READSTATUS;

- RS_success returns read was successful.
- RS_selectFailed returns slave device could not be selected

regAssign

The regAssign structure is used to construct a register assignment. It is one entry in the register assignment.

```
struct regAssign {
	unsigned	ioDriverType;
	unsigned	moduleAddress;
	unsigned	startingRegister1;
	unsigned	startingRegister2;
	unsigned	startingRegister3;
	unsigned	startingRegister4;
	unsigned	moduleType;
	unsigned	modbusStartReg1;
	unsigned	modbusStartReg2;
	unsigned	modbusStartReg3;
	unsigned	modbusStartReg4;
	};
```

- ioDriverType is the i/o module driver type
- moduleAddress is the address or group index for module
- startingRegister1 is the starting linear address of 1st group of consecutive registers mapped to module
- startingRegister2 is the starting linear address of 2nd group of registers
- startingRegister3 is the starting linear address of 3rd group of registers
- startingRegister4 is the starting linear address of 4th group of registers
- moduleType is the hardware or pseudo module type
- modbusStartReg1 is the starting Modbus register of 1st group
- modbusStartReg2 is the starting Modbus register of 2nd group
- modbusStartReg3 is the starting Modbus register of 3rd group
- modbusStartReg4 is the starting Modbus register of 4th group

routingTable

The routingTable type describes an entry in the DNP Routing Table.

Note that the DNP Routing Table is a list of routes, which are maintained in ascending order of DNP addresses.

```
typedef struct RoutingTable_type
{
      UINT16 address;                   /* station address */
      UINT16 comPort;                   /* com port interface */
      UINT16 retries;                   /* number of retries */
      UINT16 timeout;                   /* timeout in milliseconds
*/
} routingTable;
```

- adress is the DNP address.
- comPort is the serial port interface.
- retries is the number of data link retires for this table entry.
- timeout is the timeout in milliseconds.

SFTranslation

The SFTranslation structure contains Store and Forward Messaging translation information. This is used to define an address and port translation.

```
struct SFTranslation {
      unsigned portA;
      unsigned stationA;
      unsigned portB;
      unsigned stationB;
      };
```

- portA is the index of the first serial port. The index is obtained with the portIndex function.
- stationA is the station address of the first station.
- portB is the index of the second serial port. The index is obtained with the portIndex function.
- stationB is the station address of the second station.

SFTranslationStatus

The SFTranslationStatus structure contains information about a Store and Forward Translation table entry. It is used to report information about specific table entries.

```
struct SFTranslationStatus {
      unsigned index;
      unsigned code;
      };
```

- index is the location in the store and forward table to which the status code applies.
- code is the status code. It is one of SF_VALID, SF_INDEX_OUT_OF_RANGE, SF_NO_TRANSLATION, SF_PORT_OUT_OF_RANGE, SF_STATION_OUT_OF_RANGE, or SF_ALREADY_DEFINED macros.

TASKINFO

The TASKINFO type is a structure containing information about a task.

```
/* Task Information Structure */
typedef struct taskInformation_tag {
      unsigned taskID;
      unsigned priority;
      unsigned status;
      unsigned requirement;
      unsigned error;
      unsigned type;
      } TASKINFO;
```

- taskID is the identifier of the task.
- priority is the execution priority of the task.
- status is the current execution status the task. This may be one of TS_READY, TS_EXECUTING, TS_WAIT_ENVELOPE, TS_WAIT_EVENT, TS_WAIT_MESSAGE, or TS_WAIT_RESOURCE macros.
- requirement is used if the task is waiting for an event or resource. If the status field is TS_WAIT_EVENT, then requirement indicates on which event it is waiting. If the status field is TS_WAIT_RESOURCE then requirement indicates on which resource it is waiting.
- error is the task error code. This is the same value as returned by the check_error function.
- type is the task type. It will be either SYSTEM or APPLICATION.

taskInfo_tag

The taskInfo_tag structure contains start up task information.

```
struct taskInfo_tag {
        void *address;
        unsigned stack;
        unsigned identity;
        };
```

- address is the pointer to the start up routine.
- stack is the required stack size for the routine
- identity is the type of routine found (STARTUP_APPLICATION or STARTUP_SYSTEM)

timer_info

The timer_info structure contains information about a timer.

```
struct timer_info {
    unsigned time;
    unsigned interval;
    unsigned interval_remaining;
    unsigned flags;
    unsigned duty_on;
    unsigned duty_period;
    unsigned channel;
    unsigned bit;
    };
```

- time is the time remaining in the timer in ticks.
- interval is the length of a timer tick in 10ths of a second.
- interval_remaining is the time remaining in the interval count down register in 10ths of a second.
- flags is the timer type and status bits (NORMAL, PULSE TRAIN, DUTY_CYCLE, TIMEOUT, and TIMED_OUT). More than one condition may be true at any time.
- duty_on is the length of the on high portion of the square wave output. This is used only by the **pulse** function.
- duty_period is the period of the square wave output This is used only by the **pulse** function.
- channel and bit specify the digital output point. This is used by **pulse**, **pulse_train** and **timeout** functions.

VERSION

The Firmware Version Information Structure holds information about the firmware.

```
typedef struct versionInfo_tag {
       unsigned version;
       unsigned controller;
       char date[VI_DATE_SIZE + 1];
       char copyright[VI_STRING_SIZE + 1];
       } VERSION;
```

- version is the firmware version number.
- controller is target controller for the firmware.
- date is a string containing the date the firmware was created.
- copyright is a string containing Control Microsystems copyright information.

WRITESTATUS

The WRITESTATUS enumerated type indicates the status of an I^2C bus message read and may have one of the following values.

```
enum WriteStatus {
       WS_success,
       WS_selectFailed,
       WS_noAcknowledge
       };
typedef enum WriteStatus WRITESTATUS;
```

- WS_success returns write was successful
- WS_selectFailed returns slave could not be selected
- WS_noAcknowledge returns slave failed to acknowledge data

C Compiler Known Problems

The C compiler supplied with the Telepace C Tools is a product of Microtec. There are two known problems with the compiler.

Use of Initialized Static Local Variables

The compiler incorrectly allocates storage for initialized static local variables. The storage is allocated incorrectly in memory reserved for constant string data. The storage should be allocated in memory for initialized variables.

Problems Caused

A program loaded in ROM cannot modify a variable declared in this fashion.

A program loaded in RAM can modify the variable. However, the variable is in a section of program memory that the operating system expects to remain constant. Modifying the variable causes the operating system to think the program has been modified. The program continues to run correctly, but will not run again if it is stopped by the C Program Loader or if the controller is reset. The operating system detects that the program memory is corrupt and does not execute the program.

Example

The compiler generates incorrect code for the following example. Storage for the variable a is allocated in the *strings* section. It should be in the *initvars* section.

If the program is loaded in ROM, it cannot modify the variable a.

If the program is loaded in RAM, it can be run once after being written to a controller memory. All subsequent attempts to run the program will fail.

```
void main(void)
{
       static int a = 1;

       a++;
       /* other code here */
}
```

Working Around the Problem

There are two ways to work around the problem.

1. Use global variable instead of a local variable. For example:

```
static int a = 1;

void main(void)
{
       a++;
```

```
        /* other code here */
}
```

2. If the local variable is to be initialized to zero, then a non-initialized static local variable can be used. For example:

```
void main(void)
{
        static int a;

        a++;
        /* other code here */
}
```

In this example the declaration:

static int a;

is the same as the following:

static int a = 0;

The operating systems sets non-initialized variables (stored in the zerovars section) to zero before running the program.

Correction to the Problem

This problem exists with the C Compiler supplied by Microtec. It will not be corrected. Users need to work around the problem as described above.

Use of pow Function

The compiler sometimes incorrectly evaluates expressions involving the pow function with other arithmetic.

Also, a task calling the pow function requires at least 5 stack blocks. The need for more stack space by the pow function is not a compiler problem, it is simply a requirement of pow.

Problems Caused

Some arithmetic expressions involving the pow function may result in incorrect results. When testing expressions that call pow, if the result is found to be incorrect, it will be consistently incorrect for all values used by variables in the expression.

The pow function requires at least 5 stack blocks. If 4 or less stack blocks are used by the task calling pow, the controller will overflow its stack space. When the stack space overflows the behavior is unpredictable, and will likely cause the controller to reset.

Example

The compiler generates incorrect code for the following example. The result of this expression is incorrect for all values used for its variables.

```
void main(void)
{
	double a, b, c, d, e;

	a = pow(b, c) * (d + e);

	/* other code here */
}
```

Working Around the Problem

There are two ways to work around the problem.

1. To work around the problem compute the pow result on a separate line and use the result in the arithmetic expression afterwards. For example:

```
void main(void)
{
	double a, b, c, d, e, result;

	result = pow(b, c);
	a = result * (d + e);

	/* other code here */
}
```

When a task calls the pow function it requires at least 5 stack blocks. The default stack space allocated to the main task is only 4 blocks. To modify the number of stack blocks allocated to the main task refer to the section *Start-Up Function Structure* for details on editing *appstart.c*. See the function create_task to specify the stack used by other tasks.

2. The powf function may be used instead of pow where double precision is not required.

Correction to the Problem

This problem exists with the C Compiler supplied by Microtec. It will not be corrected. Users need to work around the problem as described above.