GFK-0772

GE Fanuc Manual Series 90-30

PCM C Function Library Reference Manual

GE Fanuc Automation

Programmable Control Products

PCM C Function Library Reference

Reference Manual

GFK-0772A *August 1996*

Warnings, Cautions, and Notes as Used in this Publication

Warning

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

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

Caution

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

Note

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

This document is based on information available at the time of its publication. While efforts have been made to be accurate, the information contained herein does not purport to cover all details or variations in hardware or software, nor to provide for every possible contingency in connection with installation, operation, or maintenance. Features may be described herein which are not present in all hardware and software systems. GE Fanuc Automation assumes no obligation of notice to holders of this document with respect to changes subsequently made.

GE Fanuc Automation makes no representation or warranty, expressed, implied, or statutory with respect to, and assumes no responsibility for the accuracy, completeness, sufficiency, or usefulness of the information contained herein. No warranties of merchantability or fitness for purpose shall apply.

Preface

This manual provides a complete reference to all the library functions provided in the PCM runtime libraries for the PCM C toolkit. It is written for experienced C programmers who are also familiar with the operation of Series 90 PLCs. Readers new to the C programming language or to Series 90 PLCs should familiarize themselves thoroughly with these topics before attempting to use the material in this manual. The list of publications at the end of this section contains helpful references.

Related Publications

For more information, refer to these publications:

Series 90™ -70 Programmable Controller Installation Manual (GFK-0262): This manual describes the hardware used in a Series 90-70 PLC system, and explains system setup and operation.

Logicmaster™ 90-70 Programming Software User's Manual (GFK-0263): This manual describes operation of Logicmaster 90-70 software for configuring, programming, monitoring, and controlling a Series 90-70 PLC and / or remote I / O drop.

Series 90™ -70 Programmable Controller Reference Manual (GFK-0265): This manual describes program structure and instructions for the Series 90-70 PLC.

Series 90™ -30 Programmable Controller Installation Manual (GFK-0356): This manual describes the hardware used in a Series 90-30 PLC system, and explains system setup and operation.

Logicmaster™ 90 Series 90-30/20/Micro Programming Software User's Manual (GFK-0466): This manual describes operation of Logicmaster 90-30 software for configuring, programming, monitoring, and controlling a Series 90-30 PLC.

Series 90™ -30/20/Micro Programmable Controllers Reference Manual (GFK-0467): This manual describes program structure and instructions for the Series 90-30 PLC.

C Programmer's Toolkit for Series 90t PCMs User's Manual (GFK-0771): This manual contains information about the design and construction of C language application programs for the GE Fanuc Series 90 PCM.

The C Primer. Hancock, Les, and Morris Krieger. New York: McGraw-Hill Book Co., Inc., 1982.

C: A Reference Manual. Harbison, Samuel P., and Greg L. Steele. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., Third Edition, 1988.

The C Programming Language. Kernighan, Brian W., and Dennis M. Ritchie. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., Third Edition, 1988.

Programming in C. Kochan, Stephen. Hasbrouck Heights, New Jersey: Hayden Book Co., Inc., 1983.

Learning to Program in C. Plum, Thomas. Cardiff, New Jersey: Plum Hall, Inc., 1983.

We Welcome Your Comments and Suggestions

At GE Fanuc Automation, we strive to produce quality technical documentation. After you have used this manual, please take a few moments to complete and return the Reader's Comment Card located on the next page.

Henry Konat
Senior Technical Writer

Contents

Chapter 1 PCM C Functions

This chapter provides a complete reference to all the library functions provided with the PCM C toolkit. Two categories of functions are included: VTOS operating system services and Series 90 PLC service request application program interface (PLC API) services. All the functions are listed in alphabetical order.

Content of this Manual

The reference for each library function contains these items:

■ Usage

This section provides the calling format or formats for the function. It includes the header file where the function prototype is defined plus the type specifications of the function return value and all calling parameters. Function names and type names appear in **`this typeface`**, while formal parameter names appear in *`this typeface`*.

Some functions have a variable number of parameters (for example, **`Open_dev`** and its relatives). For these functions, all valid calling formats are defined.

■ Description

The purpose of the function and all its parameters are defined in this section. Valid ranges of parameter values are also defined.

■ Return Value

The value or values returned by the function, including error codes, are defined in this section.

■ See Also

This section contains a list of related functions. Functions which are used together (for example, **`Link_sem`**, **`Unblock_sem`**, **`Block_sem`**, and **`Unlink_sem`**) and functions with complementary purposes (for example, **`Suspend_task`** and **`Resume_task`**) reference each other here.

■ Example

An example of C language code which uses the function is provided here. Examples range from small code fragments to complete programs.

VTOS Service Functions By Category

This section summarizes VTOS services by grouping them in categories of related services. For full details on all VTOS services, see the following pages of this manual.

Task Management Functions:

Function Name	Purpose
Get_task_id	Returns the task ID value of the calling task.
Init_task	Executes a VTOS application program or driver as a task.
Process_env	Starts a PCM task using a saved environment block.
Resume_task	Allows a suspended task to execute.
Set_std_device	Sets a standard I / O channel for a task.
Suspend_task	Prevents a task from executing.
Terminate_task	Kills a task permanently and frees its resources.
Test_task	Returns the set of active tasks.
Wait_task	Waits for a specified task to terminate.

Event Flag Functions:

Function Name	Purpose
Iset_ef	Sets one or more local event flags from an interrupt or communication timer service routine.
Iset_gef	Sets one or more global event flags from an interrupt or communication timer service routine.
Reset_ef	Resets one or more local event flags.
Reset_gef	Resets one or more global event flags.
Set_ef	Sets one or more local event flags.
Set_gef	Sets one or more global event flags.
Test_ef	Tests one or more local event flags.
Test_gef	Tests one or more global event flags.
Wait_ef	Waits for one or more local event flags to be set.
Wait_gef	Waits for one or more global event flags to be set.

Asynchronous Trap Functions:

Function Name	Purpose
Disable_asts	Prevents the calling task from processing ASTs.
Enable_asts	Permits the calling task to process ASTs.
Post_ast	Sends an asynchronous trap to a specified task.
Wait_ast	Suspends execution of the calling task until an AST is received.

Semaphore Functions:

Function Name	Purpose
Block_sem	Check whether a semaphore is open; wait if not.
Link_sem	Link the calling task to a named semaphore; create one if it is not found.
Unblock_sem	Release a semaphore and activate the first waiting task.
Unlink_sem	Unlink the calling task from a semaphore.

Time of Day Clock Functions:

Function Name	Purpose
Elapse	Returns a count of milliseconds since the last time its count was reset.
Get_date	Returns the current date.
Get_time	Returns the current time of day.

Timer Functions:

Function Name	Purpose
Cancel_timer	Stops a timer and undefines it.
Start_timer	Defines a timer and starts it counting from zero.
Wait_time	Suspends execution of the calling task for the specified time.

Communication Timer Functions:

Function Name	Purpose
Alloc_com_timer	Allocatesa communication timer to the calling task.
Cancel_com_timer	Stops a communication timer.
Dealloc_com_timer	De-allocates a communication timer.
Start_com_timer	Starts a previously allocated communication timer.

Memory Management Functions:

Function Name	Purpose
Get_best_buff	Allocatesmemory from the mallest free memory block that is at least as large as the requested size.
Get_buff	Allocatesmemory from the first free memory block which is at least as large as the requested size.
Get_dp_buff	Allocatesmemory from free VMEbus dual ported RAM in a Series 90-70 PCM.
Get_mem_lim	Returns the starting address of a memory block reserved for applicationprograms.
Max_avail_buff	Returns the size in bytes of the largest free memory block.
Max_avail_mem	Returns the total number of bytes in free memory.
Reserve_dp_buff	Reserves a specified block of VMEbus dual ported RAM in a Series 90-70 PCM.
Return_buff	Returns the specified memory buffer to the free memory pool.
Return_dp_buff	Returns the specified block of VMEbus dual ported RAM to the free memory in a Series 90-70 PCM.

Memory Module Functions:

Function Name	Purpose
Get_mod	Returns the address of a named memory module.

Device I/O Functions:

Function Name	Purpose
Abort_dev	Abortsone or more I / O operations on a previously opened I / O channel.
Close_dev	Closes a previously opened I / O channel.
Devctl_dev	Performs a specified control operation on a named device.
Ioctl_dev	Performs a specified control operation on a previously opened I / Ochannel.
Open_dev	Opens a channel on a named I / O device.
Read_dev	Returns input data from a previously opened I / O channel.
Seek_dev	Positions the data pointer of a previously opened I / O channel to a specified location.
Special_dev	Performs a special operation on a previously opened I / O channel.
Write_dev	Sends output to a previously opened I / O channel.

Device Driver Support Functions:

Function Name	Purpose
Get_next_block	Returns device argument blocks to a VTOS device driver.
Install_dev	Installs a VTOS device driver.
Install_isr	Installs a VTOS interrupt service routine.
Notify_task	Notifies a VTOS task when a device operation completes.

Miscellaneous Functions:

Function Name	Purpose
Define_led	Defines the function of one of the programmable light–emitting diodes (LEDs).
Get_board_id	Returns the PCM hardware type.
Get_pcm_rev	Returns the revision number of VTOS.
Set_dbd_ctl	Sets the Series 90–70 PCM daughterboard control register.
Set_led	Sets the state of one of the LEDs.
Where_am_i	Returns the PLC rack / slot location of the PCM.

VME Functions:

Function Name	Purpose
Set_vme_ctl	Sets the VMEbus access parameters in a Series 90–70 PCM.
Vme_clear_lcl_sem	Clear a semaphore in VMEbus dual ported RAM of the local Series 90–70 PCM.
Vme_read	Read VMEbus dual ported RAM in a different module.
Vme_test_and_set	Acquire a semaphore in VMEbus dual ported RAM of a different module.
Vme_test_lcl_sem	Acquire a semaphore in VMEbus dual ported RAM of the local Series 90–70 PCM.
Vme_write	Write VMEbus dual ported RAM in a different module.

PCM 712 Functions:

Function Name	Purpose
Send_vme_interrupt	AssertsVME IRQ7 interrupt request from a PCM712.
Set_local_date	Sets the current date maintained by a PCM712.
Set_local_time	Sets the time of day maintained by a PCM712.

PLC Service Request Interface API Service Functions By Category

This section summarizes PLC API services by grouping them in categories of related services. For full details on all PLC API services, see the following pages of this manual.

Managing API Services:

Function Name	Purpose
api_initialize	Initialize data used by PLC API services.
configure_comm_link	Specifies the communication path to the PLC CPU.
establish_comm_session	Establishes communication with the PLC CPU.
release_request_id	Frees a request identifier returned by a previous nowait API request.
reqstatus	Returns the current status of a pending nowait API request.
terminate_comm_session	Terminates communication with the PLC CPU.

PLC Hardware Type, Configuration, and Status Information:

Function Name	Purpose
get_cpu_type_rev get_cpu_type_rev_nowait	Return the PLC CPU hardware type and firmware revision.
get_memtype_sizes get_memtype_sizes_nowait	Return the sizes of user–configurable PLC memory types.
chg_priv_level chg_priv_level_nowait	Change the PLC access privilege level of the calling task.
update_plc_status update_plc_status_nowait	Update PLC status data in the global structure `plc_status_info`.

PLC Program and Configuration Checksum Data:

Function Name	Purpose
get_prgm_info get_prgm_info_nowait	Return the PLC program checksums.
get_config_info get_config_info_nowait	Return the PLC configuration data checksums.

Reading PLC Data References:

Function Name	Purpose
read_sysmem read_sysmem_nowait	Read up to 2048 bytes of a single PLC reference type.
read_prgmdata read_prgmdata_nowait	Read up to 2048 bytes of Series 90–70 %P data.
read_localdata read_localdata_nowait	Read up to 2048 bytes of Series 90–70 %L data.

Writing PLC Data References:

Function Name	Purpose
write_sysmem write_sysmem_nowait	Write up to 2048 bytes of a single PLC reference type.
write_prgmdata write_prgmdata_nowait	Write up to 2048 bytes of Series 90–70 %P data.
write_localdata write_localdata_nowait	Write up to 2048 bytes of Series 90–70 %L data.

Controlling PLC Operation:

Function Name	Purpose
start_plc start_plc_nowait	Set the PLC state to RUN mode.
start_plc_noio start_plc_noio_nowait	Set the PLC state to RUN mode with outputs disabled. (Series 90–70 PLC CPU request only)
stop_plc stop_plc_nowait	Set the PLC state to STOP mode.

Reading Mixed PLC Data References:

Function Name	Purpose
establish_mixed_memory establish_mixed_memory_nowait	Establish a mixed memory shopping list for subsequent read_mixed_memory or read_mixed_memory_nowait calls.
read_mixed_memory read_mixed_memory_nowait	Get the mixed memory data previously specified by an establish_mixed_memory or establish_mixed_memory_nowait call.
cancel_mixed_memory cancel_mixed_memory_nowait	Cancel the mixed memory shopping list previously specified by an establish_mixed_memory or establish_mixed_memory_nowait call.

Reading and Clearing PLC and I/O Faults:

Function Name	Purpose
clr_plc_fault_tbl clr_plc_fault_tbl_nowait	Clear the entire PLC fault table.
clr_io_fault_tbl clr_io_fault_tbl_nowait	Clear the entire I / O fault table.
chk_genius_bus chk_genius_bus_nowait	Used to determine whether the specified Genius bus on the module in the specified rack and slot has a faulted device. (Series 90–70 PLC CPU request only)
chk_genius_device chk_genius_device_nowait	Used to determine whether the specified Genius device at the specified bus, rack, and slot address is faulted. (Series 90–70 PLC CPU request only)
get_one_rackfaults get_one_rackfaults_nowait	Return all the system fault bits for the specified PLC rack. (Series 90–70 PLC CPU request only)
get_rack_slot_faults get_rack_slot_faults_nowait	Used to determine which slot or slots, if any, in a specified rack have faulted modules. (Series 90–70 PLC CPU request only)
read_plc_fault_tbl read_plc_fault_tbl_nowait	Read the entire PLC fault table.
read_io_fault_tbl read_io_fault_tbl_nowait	Read the entire I / O fault table.

Reading and Setting the PLC Time of Day Clock:

Function Name	Purpose
read_date read_date_nowait	Return the current date from the PLC time of day clock.
read_time read_time_nowait	Return the current time from the PLC time of day clock.
read_timedate read_timedate_nowait	Return the current time and date from the PLC time of day clock in a single operation.
set_date set_date_nowait	Set the date in the PLC time of day clock.
set_time set_time_nowait	Set the time in the PLC time of day clock.
set_timedate set_timedate_nowait	Set the time and date in both the PLC and PCM time of day clocks in a single operation.

Abort_dev

Usage

```
#include <vtos.h>

word Abort_dev ( channel_handle, operation, notify_flag );
word channel_handle;
word operation;
word notify_flag;
```

Description

This function is used to abort I/O operations which are currently in progress on a specified channel. **Abort_dev** can be used to selectively terminate all operations of a given type, such as reads or writes, or it can be used to terminate all outstanding operations for the specified channel. The *channel_handle* parameter must contain a value which was returned by **Open_dev**. It specifies the channel for the operation being aborted. The *operation* parameter specifies the type of operation to be aborted. It must contain a value from this table.

operation Value	Type of Operation Aborted
ABORT_ALL	All device operations.
ABORT_OPEN	**Open_dev**
ABORT_CLOSE	**Close_dev**
ABORT_READ	**Read_dev**
ABORT_WRITE	**Write_dev**
ABORT_SEEK	**Seek_dev**
ABORT_IOCTL	**Ioctl_dev**
ABORT_SPECIAL	**Special_dev**
ABORT_DEVCTL	**Devctl_dev**

Abort_dev is most commonly used to abort **Read_dev** and **Write_dev** operations.

The *notify_flag* parameter determines whether the task which initiated the I/O operation will be notified when the abort operation has completed. If its value is non-zero, the task will be notified in the manner originally specified: by unblocking the task (WAIT); setting an event flag (EVENT_NOTIFY); or posting an AST (AST_NOTIFY). An ABORTED error status will be reported for each aborted I/O operation.

When a **Read_dev** or **Write_dev** operation is aborted, the number of characters transferred before the abort is reported as shown in this table.

	notify_code **of Aborted Operation**		
	WAIT	**EVENT_NOTIFY**	**AST_NOTIFY**
Result structure type	None	**device_result**	**ast_blk**
Number of characters transferred is in:	Function return value	**ioreturn** member	**arg2** member
ABORT status is in:	**_VTOS_error**	**iostatus** member	**arg1** member

If *notify_flag* is set to WAIT, the initiating task will not be notified.

Caution

If one task has initiated a WAIT mode I/O operation, and a different task aborts the operation with *notify_flag* set to WAIT, the task which is waiting for the I/O operation to complete will never resume execution.

■ Return Value

SUCCESS is returned when there are no errors. If *channel_handle* is invalid or is not currently performing the specified type of I/O operation, IO_FAILED is returned, and **_VTOS_error** contains **BAD_HANDLE**.

■ See Also

Read_dev, Seek_dev, Write_dev

■ Example

```
#include <vtos.h>

word handle, task;
device_result evt_result;
byte buffer [130];

task = Get_task_id ();
handle = Open_dev ( "COM1:", READ_MODE, WAIT, task );

Read_dev ( handle, buffer, 128, EVENT_NOTIFY, task, EF_00,
                   (device_result far* )&evt_result );

Abort_dev ( handle, ABORT_READ, TRUE );

if ( evt_result.ioreturn != 0 ) {
    /* Some characters were received before the abort. */
}
```

This example opens PCM serial port one for reading and then starts a read operation in EVENT_NOTIFY mode. **Abort_dev** is called to abort the read operation, and the number of characters received before the abort is checked.

Alloc_com_timer

Usage

```
#include <vtos.h>

word Alloc_com_timer ( void );
```

Description

This function allocates one of four PCM communication timers. Communication timers use PCM hardware interrupts and timers; they involve less processing overhead than general purpose timers. There are no function parameters.

VTOS itself uses communication timers for CCM and serial communication. Timers may or may not be available for C applications.

Return Value

When the call succeeds, a non-zero timer handle is returned. If no timers are available, zero is returned, and **`_VTOS_error`** contains NO_TIMERS.

Note

Communication timer handles and general purpose timer handles are **not** interchangeable.

See Also

`Cancel_com_timer, Dealloc_com_timer, Start_com_timer`

Example

```
#include <vtos.h>

word com_timer_handle;
com_timer_handle = Alloc_com_timer ();
```

api_initialize

Usage

```
#include <session.h>

void api_initialize ( void );
```

Description

This function must be called before any other PLC API interface call. It performs all the initialization required for communication with the PLC CPU.

Return Value

None.

See Also

`configure_comm_link, establish_comm_session, terminate_comm_session`

Example

```
#include <session.h>

BYTE sesn_id;
REQSTAT status;

main ()
{

    api_initialize ();
    if ( configure_comm_link ( "S90BP", NULL ) ) {
       status = establish_comm_session ( "#7", &sesn_id );
       if ( status ) {
           /* An error occurred -- the least significant byte of */
           /* status contains the major error code, and the most */
           /* significant byte contains the minor error code.    */
       } else {
           /* Make PLC API service request calls.                */
       status = terminate_comm_session ( sesn_id );
       }
    }
}
```

This example initializes the PLC service request application program interface and opens a communication session with the PLC CPU.

Caution

The `device_id` string parameter for `establish_comm_session` specifies a service point on the CPU: device. The same service point number must <u>not</u> be used in an `Open_dev` call.

Block_sem

Usage

```
#include <vtos.h>

void Block_sem ( semaphore_handle );
word semaphore_handle;
```

Description

This function provides controlled access to PCM resources, such as memory modules or other data objects in PCM memory, which are shared by two or more tasks. When the shared resource is free, the **Block_sem** call returns immediately. But if the resource is in use by another task, the call does not return until the resource is free. The *semaphore_handle* must be a value returned by a previous, successful call to **Link_sem**.

Unblock_sem must be called after each **Block_sem** call. A call to **Unblock_sem** must also be made after the task calls **Link_sem** to link to the semaphore.

When a task is forced to wait at a semaphore, its priority is compared to the priority of the task which currently controls the same semaphore. If the waiting task has higher priority, the controlling task is temporarily promoted to the priority of the waiting task. This allows the waiting task to gain control of the semaphore as quickly as possible. If more than one task is waiting at the semaphore, the priority of the controlling task is promoted to that of the highest priority waiting task.

Caution

Attempting to block on more than one semaphore at the same time is likely to cause the calling task to deadlock. Since VTOS device operations make extensive use of semaphores, applications should never make a VTOS or PLC API function call while blocked on a semaphore.

For more information on semaphores, see chapter 7, *Multitasking*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

Return Value

None.

See Also

Link_sem, Unblock_sem, Unlink_sem

Example

```
#include <vtos.h>

word sem_handle;

sem_handle = Link_sem ( ”MY_SEM” );
/* Access or modify the protected data. */
Unblock_sem ( sem_handle );

Block_sem ( sem_handle );
/* Access or modify the protected data. */
Unblock_sem ( sem_handle );
```

Cancel_com_timer

Usage

```
#include <vtos.h>

word Cancel_com_timer ( com_timer_handle);
word com_timer_handle;
```

Description

This function is used to stop (cancel) a communication timer which was previously started by **Start_com_timer**. The *com_timer_handle* is a handle returned by a previous, successful call to **Alloc_com_timer**. The function has no effect if *com_timer_handle* is invalid or the specified timer is not running.

Return Value

None.

See Also

Alloc_com_timer, Dealloc_com_timer, Start_com_timer

Example

See **Start_com_timer**.

cancel_mixed_memor y

Usage

```
#include <mxread.h>

REQSTAT cancel_mixed_memory ( session_id, list_id );
BYTE session_id;
BYTE list_id;
```

Description

This function cancels *list_id*, a mixed memory "shopping list" which was previously defined by a successful call to **establish_mixed_memory** or **establish_mixed_memory_nowait**. Since at most two lists may be active at any time, this function or **cancel_mixed_memory_nowait** should be used to cancel an obsolete list before defining a new one. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
CONN_ID_NOT_FOUND	REQUEST_ERROR	The data list specified by *list_id* has not been established.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_ comm_link**, and **establish_comm_ session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.

See Also

cancel_mixed_memory_nowait, establish_mixed_memory, establish_mixed_memory_nowait, read_mixed_memory, read_mixed_memory_nowait

Example

See **establish_mixed_memory**.

cancel_mixed_memor y_nowait

Usage

```
#include <mxreadnw.h>

REQID cancel_mixed_memory_nowait ( session_id, list_id );
BYTE  session_id;
BYTE  list_id;
```

Description

See **cancel_mixed_memory**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
CONN_ID_NOT_FOUND	REQUEST_ERROR	The data list specified by *list_id* has not been established.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_ comm_link**, and **establish_comm_ session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.

See Also

cancel_mixed_memory, establish_mixed_memory, establish_mixed_memory_nowait, read_mixed_memory, read_mixed_memory_nowait, reqstatus

Example

See **establish_mixed_memory_nowait.**

Cancel_timer

Usage

```
#include <vtos.h>

word Cancel_timer ( timer_handle );
word timer_handle;
```

Description

This function is used to stop (cancel) a general purpose timer which was previously started by **Start_timer**. The *timer_handle* is the handle returned by **Start_timer**. The function has no effect if *timer_handle* is invalid.

Return Value

SUCCESS is returned when there are no errors. FAILURE is returned when *timer_handle* is invalid, and **_VTOS_error** contains **BAD_TIMER**.

See Also

Start_timer

Example

See **Start_timer**.

chg_priv_level

Usage

```
#include <utils.h>

REQSTAT chg_priv_level ( session_id, user_password );
BYTE      session_id;
char far* user_password;
```

Description

This function enables the requesting process to change its PLC access privilege level. The address in *user_password* must point to a NUL terminated ASCII string which contains a valid PLC password. The string length, not including the NUL character, is limited to 7 characters. PLC passwords are case sensitive; uppercase letters **only** should be used. If the password is valid, the requester's privilege is changed to the highest level assigned to the password. If PLC passwords have been inactivated or disabled by the Logicmaster 90 configuration software package, or if no passwords have been assigned, *user_password* must point to an ASCII NUL character. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. If an invalid password is specified or passwords are disabled in the PLC CPU, the return value indicates failure.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PASSWORD	REQUEST_ERROR	The *user_password* is not a correct password for any PLC access privilege level, or passwords have been inactivated or disabled.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

update_plc_status

Example

```
#include <utils.h>

REQSTAT status;
char password [] = ”MYPWORD”;

status = chg_priv_level ( session_id, password );

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* at a new PLC access privilege level */
}
```

This example assumes that MYPWORD has been assigned by Logicmaster 90 as a password. It uses a WAIT mode request to change the PLC access privilege level.

chg_priv_level_nowait

Usage

```
#include <utilsnw.h>

REQID chg_priv_level_nowait ( session_id, user_password );
BYTE      session_id;
char far* user_password;
```

Description

See **chg_priv_level**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PASSWORD	REQUEST_ERROR	The *user_password* is not a correct password for any PLC access privilege level, or passwords have been inactivated or disabled.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

reqstatus, update_plc_status

Example

```
#include <utilsnw.h>

REQID request_id;
REQSTAT status;
char password [] = ”MYPWORD”;

request_id = chg_priv_level_nowait ( session_id, password );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* at a new PLC access privilege level */
}
```

This example assumes that MYPWORD has been assigned by Logicmaster 90 as a password. It uses a NOWAIT mode request to change the PLC access privilege level.

chk_genius_bus

Usage

```
#include <faults.h>

REQSTAT chk_genius_bus ( session_id, rack_num, slot_num,
                         bus_num, bus_faulted );
BYTE         session_id;
BYTE         rack_num;
BYTE         slot_num;
BYTE         bus_num;
BOOLEAN far* bus_faulted;
```

Description

This function allows the user to determine if a particular GENIUS bus, specified by the rack / slot address of a Series 90-70 GENIUS Bus Controller module (in *rack_num* and *slot_num*, respectively), and by the controller bus number (in *bus_num*), is faulted. This request is valid only for Series 90-70 PLCs. Valid rack numbers are 0 through 7, valid slot numbers are 0 through 9, and valid bus numbers are 0 and 1. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. When the request has completed successfully, the BOOLEAN variable whose address is specified in *bus_faulted* will contain TRUE if the specified bus is faulted, and FALSE otherwise.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *rack_num*, *slot_num*, or *bus_num* out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus_nowait, chk_genius_device, chk_genius_device_nowait, get_one_rackfaults, get_one_rackfaults_nowait, get_rack_slot_faults, get_rack_slot_faults_nowait

Example

```
#include <faults.h>

REQSTAT status;
BOOLEAN gbus_fltd;

status = chk_genius_bus ( session_id, 0, 3, 1, &gbus_fltd );

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else if ( gbus_flted ) {
    /* the specified Genius bus is faulted */
} else {
    /* the bus is not faulted */
}
```

This example uses a WAIT mode request to check bus one of the Genius Bus Controller in slot three of PLC rack zero to determine whether it is faulted.

chk_genius_bus_nowait

Usage

```
#include <faultsnw.h>

REQID chk_genius_bus_nowait ( session_id, rack_num, slot_num,
                              bus_num, bus_faulted );
BYTE          session_id;
BYTE          rack_num;
BYTE          slot_num;
BYTE          bus_num;
BOOLEAN far*  bus_faulted;
```

Description

See **chk_genius_bus**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *rack_num*, *slot_num*, or *bus_num* out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus, chk_genius_device, chk_genius_device_nowait, get_one_rackfaults, get_one_rackfaults_nowait, get_rack_slot_faults, get_rack_slot_faults_nowait, reqstatus

Example

```
#include <faultsnw.h>

REQID request_id;
REQSTAT status;
BOOLEAN gb_fltd;

request_id = chk_genius_bus_nowait ( session_id, 0, 3, 1, &gb_fltd );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else if ( gb_flted ) {
    /* the specified Genius bus is faulted */
} else {
    /* the bus is not faulted */
}
```

This example uses a NOWAIT mode request to check bus one of the Genius Bus Controller in slot three of PLC rack zero to determine whether it is faulted.

chk_genius_device

Usage

```
#include <faults.h>

REQSTAT chk_genius_device ( session_id, rack_num, slot_num,
                            bus_num, device_num, device_faulted );
BYTE          session_id;
BYTE          rack_num;
BYTE          slot_num;
BYTE          bus_num;
BYTE          device_num;
BOOLEAN far*  device_faulted;
```

Description

This function allows the user to determine if a particular Series 90-70 GENIUS device (for example, a genius block or Hand Held Monitor) is faulted. This request is valid only for Series 90-70 PLCs. The device is specified by the rack / slot address of a Series 90-70 GENIUS Bus Controller module (in *rack_num* and *slot_num*, respectively), the controller bus number (in *bus_num*), and the device number (in *device_num*) on the bus. Valid rack numbers are 0 through 7, valid slot numbers are 0 through 9, valid bus numbers are 0 and 1, and valid device numbers are 0 through 31. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. When the request has completed successfully, the BOOLEAN variable whose address is specified in *device_faulted* will contain TRUE if the specified bus is faulted, and FALSE otherwise.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *rack_num*, *slot_num*, *bus_num*, or *device_num* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus, chk_genius_bus_nowait, chk_genius_device_nowait, get_one_rackfaults, get_one_rackfaults_nowait, get_rack_slot_faults, get_rack_slot_faults_nowait

Example

```
#include <faults.h>

REQSTAT status;
BOOLEAN gdev_fltd;

status = chk_genius_device ( session_id, 0, 3, 1, 30, &gdev_fltd );

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else if ( gbus_flted ) {
    /* the specified Genius device is faulted */
} else {
    /* the device is not faulted */
}
```

This example uses a WAIT mode request to check device 30 on bus one of the Genius Bus Controller in slot three of PLC rack zero to determine whether it is faulted.

chk_genius_device_nowait

Usage

```
#include <faultsnw.h>

REQID chk_genius_device_nowait ( session_id, rack_num,
                                 slot_num, bus_num,
                                 device_num, device_faulted );
BYTE          session_id;
BYTE          rack_num;
BYTE          slot_num;
BYTE          bus_num;
BYTE          device_num;
BOOLEAN far*  device_faulted;
```

Description

See **chk_genius_device**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *rack_num*, *slot_num*, *bus_num*, or *device_num* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus, chk_genius_bus_nowait, chk_genius_device, get_one_rackfaults, get_one_rackfaults_nowait, get_rack_slot_faults, get_rack_slot_faults_nowait, reqstatus

Example

```
#include <faultsnw.h>

REQID request_id;
REQSTAT status;
BOOLEAN gdev_fltd;

request_id = chk_genius_device_nowait( session_id, 0, 3, 1, 30, &gdev_fltd );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else if ( gbus_flted ) {
    /* the specified Genius device is faulted */
} else {
    /* the device is not faulted */
}
```

This example uses a NOWAIT mode request to check device 30 on bus one of the Genius Bus Controller in slot three of PLC rack zero to determine whether it is faulted.

Close_dev

Usage

```
#include <vtos.h>

word Close_dev ( device_handle,  notify_code,
                 task_id [, <nowait options>] );
word device_handle;
word notify_code;
word task_id;
```

where **<***nowait options***>** depend on the value of *notify_code:*

```
word Close_dev ( device_handle, WAIT, task_id );

word Close_dev ( device_handle, EVENT_NOTIFY, task_id,
                 local_ef_mask, (device_result far*)result_ptr );
word               local_ef_mask;
device_result far* result_ptr;

word Close_dev ( device_handle, AST_NOTIFY, task_id,
                 ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

This function is used to close a device or file which was previously opened by a successful call to **Open_dev**; *device_handle* is the handle returned by **Open_dev**. The *task_id* is the task number returned by **Get_task_id**.

The *notify_code* specifies the method used to notify the calling task that the operation has completed; its value may be WAIT, EVENT_NOTIFY, or AST_NOTIFY. When WAIT is used, there are no *nowait options*, and the return from **Close_dev** is delayed until the operation completes. The other *notify_code* values cause the function to return immediately, allowing the calling task to continue execution.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Close_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a **far** pointer because its type is not specified by the function prototype in **vtos.h**. If the call succeeds, the **ioreturn** member of the structure at *result_ptr* contains SUCCESS and the **iostatus** member is undefined; when a failure occurs, **ioreturn** contains IO_FAILED, and **iostatus** contains an error status code. For a discussion of asynchronous I/O using event flags, see chapter 6, *PCM Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle* contains a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine*, it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**; it is undefined if no *ast_handle* was specified. If the call succeeds, the **arg2** member of the **ast_blk** contains SUCCESS and the **arg1** member is undefined; when a failure occurs, **arg2** contains IO_FAILED, and **arg1** contains an error status code. For a discussion of asynchronous I/O using AST functions, see chapter 6, *PCM Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

■ Return Value

In WAIT mode, SUCCESS is returned when there are no errors. When an error occurs, IO_FAILED is returned; a status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function is undefined and should be ignored. The actual return and status values are available in **device_result** and **ast_blk** structures, respectively.

For all modes, the return and status variables contain values from this table.

Return Value	Status Value	Completion Status
SUCCESS	Undefined	The device was successfully closed.
IO_FAILED	BAD_HANDLE	An invalid *device_handle* was specified.
	ABORTED	The operation was aborted before completion.

■ See Also

Get_task_id, Open_dev, Reset_ef, Test_ef, Wait_ef

■ Example

```
#include <vtos.h>
#include <dos.h>
#define AST_CLOSE 27

word close_wait_error, close_ef_error, close_ast_error, close_ast_done;

void far close_ast_func ( ast_blk far* p )
{
    if ( p->handle == AST_CLOSE ) {
       close_ast_done = 1;
       if ( p->arg2 != SUCCESS ) {
          /* There was a problem. */
          close_ast_error = p->arg1;
       }
    }
}
```

```
void main ()
{
    word h1, h2, h3, task_id, status;
    device_result evt_result;
    task_id = Get_task_id ();

    h1 = Open_dev ( "Com1:", WRITE_MODE, WAIT, task_id );
    h2 = Open_dev ( "Com2:", WRITE_MODE, WAIT, task_id );
    h3 = Open_dev ( "CPU:%R1", WRITE_MODE, WAIT, task_id );
/*
 * Do some I/O operations.
 */
    Reset_ef ( EF_01 );
    close_wait_error = close_ef_error =
       close_ast_done = close_ast_error = 0;

    Abort_dev( h1, ABORT_ALL, WAIT );
    Abort_dev( h2, ABORT_ALL, WAIT );

    Close_dev ( h1, EVENT_NOTIFY, task_id,
                 EF_01, (device_result far* )&evt_result );

    Close_dev ( h2, AST_NOTIFY, task_id,
                 close_ast_func, AST_CLOSE );

    status = Close_dev ( h3, WAIT, task_id );
    if ( status != SUCCESS ) {
       /* There was a problem. */
       close_wait_error = _VTOS_error;
    }

    Wait_ef ( EF_01 );

    if ( evt_result.ioreturn != SUCCESS ) {
       /* There was a problem */
       close_ef_error = evt_result.iostatus;
    }

    _disable ();
     if ( !close_ast_done ) {
       Wait_ast ();
     }
    _enable ();
}
```

In this example, **main** opens three I/O channels on the PCM devices **COM1:**, **COM2:**, and **CPU:**. After some I/O transfer operations, which are not shown here, I/O operations on the serial ports are aborted to assure that any pending nowait read or write operations are stopped. Then, the channels are closed using EVENT_NOTIFY, AST_NOTIFY, and WAIT modes, respectively. The program makes the EVENT_NOTIFY and AST_NOTIFY requests first, followed by the WAIT request. After the WAIT request has completed, the program waits for completion of the EVENT_NOTIFY and AST_NOTIFY requests. When a problem occurs, **close_wait_error**, **close_ef_error**, and/or **close_ast_error** will contain an error code.

Note that a call to **_disable** is used to disable maskable PCM interrupts before **close_ast_done** is tested. This prevents the AST from being processed between the test and the **Wait_ast** call. If the AST is processed before **Wait_ast** is called, the call never returns. The **_enable** function must be called whenever **_disable** is used.

clr_io_fault_tbl

Usage

```
#include <clrflt.h>

REQSTAT clr_io_fault_tbl ( session_id );
BYTE session_id;
```

Description

This function permits the user to clear all existing faults in the I/O Fault Table of the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl_nowait, clr_plc_fault_tbl, clr_plc_fault_tbl_nowait, read_io_fault_tbl, read_io_fault_tbl_nowait, read_plc_fault_tbl, read_plc_fault_tbl_nowait

Example

```
#include <clrflt.h>

REQSTAT status;

status = clr_io_fault_tbl ( session_id );

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the I/O fault table was cleared */
}
```

This example uses a WAIT mode request to clear the I/O fault table in the PLC CPU.

clr_io_fault_tbl_nowait

Usage

```
#include <clrfltnw.h>

REQID clr_io_fault_tbl_nowait ( session_id );
BYTE session_id;
```

Description

See **clr_io_fault_tbl**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl, clr_plc_fault_tbl, clr_plc_fault_tbl_nowait, read_io_fault_tbl, read_io_fault_tbl_nowait, read_plc_fault_tbl, read_plc_fault_tbl_nowait, reqstat

reqstatus

Example

```
#include <clrfltnw.h>

REQID request_id;
REQSTAT status;

request_id = clr_io_fault_tbl_nowait ( session_id );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the I/O fault table was cleared */
}
```

This example uses a NOWAIT mode request to clear the I/O fault table in the PLC CPU.

clr_plc_fault_tbl

Usage

```
#include <clrflt.h>

REQSTAT clr_plc_fault_tbl ( session_id );
BYTE session_id;
```

Description

This function permits the user to clear all existing faults in the PLC Fault Table of the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl, clr_io_fault_tbl_nowait, clr_plc_fault_tbl_nowait, read_io_fault_tbl, read_io_fault_tbl_nowait, read_plc_fault_tbl, read_plc_fault_tbl_nowait

Example

```
#include <clrflt.h>

REQSTAT status;

status = clr_plc_fault_tbl ( session_id );

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC fault table was cleared */
}
```

This example uses a WAIT mode request to clear the PLC fault table in the PLC CPU.

clr_plc_fault_tbl_nowait

Usage

```
#include <clrfltnw.h>

REQID clr_plc_fault_tbl_nowait ( session_id );
BYTE session_id;
```

Description

See **clr_plc_fault_tbl**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl, clr_io_fault_tbl_nowait, clr_plc_fault_tbl, read_io_fault_tbl, read_io_fault_tbl_nowait, read_plc_fault_tbl, read_plc_fault_tbl_nowait, reqstatus

Example

```
#include <clrfltnw.h>

REQID request_id;
REQSTAT status;

request_id = clr_plc_fault_tbl_nowait ( session_id );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC fault table was cleared */
}
```

This example uses a NOWAIT mode request to clear the PLC fault table in the PLC CPU.

configure_comm_link

Usage

```
#include <session.h>

BOOLEAN configure_comm_link ( comm_id_string, config_data );
char far* comm_id_string;
void far* config_data;
```

Description

This function must be called after **api_initialize**. It is called to identify the link which will be used to communicate with the PLC CPU. The *comm_id_string* parameter must contain the address of the character string **"S90BP"**. The address in the *config_data* parameter is ignored by the PCM implementation of the PLC API.

The call to **establish_comm_session** must occur after **configure_comm_link**.

Return Value

TRUE is returned if **api_initialize** has already been called, **establish_comm_session** has not been called, and *comm_id_string* points to the value **"S90BP"**. If any of these conditions is not met, FALSE is returned.

See Also

api_initialize, establish_comm_session

Example

See **api_initialize**.

Dealloc_com_timer

Usage

```
#include <vtos.h>

void Dealloc_com_timer ( com_timer_handle);
word com_timer_handle;
```

Description

This function deallocates a communication timer which was previously allocated by **Alloc_com_timer.** The *com_timer_handle* is the communication timer handle returned by **Alloc_com_timer**. The function has no effect if *com_timer_handle* is invalid.

Dealloc_com_timer is rarely used. A task's communication timers are automatically deallocated if the task terminates.

Return Value

None.

See Also

Alloc_com_timer, Cancel_com_timer, Start_com_timer

Example

```
#include <vtos.h>

word com_timer_handle;
com_timer_handle = Alloc_com_timer ();
/*
 * use the timer
 */
Dealloc_com_timer ( com_timer_handle );
```

Define_led

Usage

```
#include <vtos.h>

word Define_led ( led_number, led_definition );
word led_number;
word led_definition;
```

Description

This function has two purposes. It defines the communication events which will cause one of two light emitting diodes (LEDs) on the PCM to flash. It is also used to specify which PCM task will be permitted to control one of the LEDs with the **Set_led** function.

The top LED reports the operational status of the PCM and is not programmable. LED 1, the center LED, and LED 2, the bottom LED, may be programmed by **Define_led**. The *led_number* must contain 1 or 2, to specify LED 1 or LED 2, respectively.

When **Define_led** is used to define PCM communication events which will flash an LED, the least significant byte of *led_definition* contains one or more event definitions from this table, OR-ed together.

Event Definition	Description
COM1_XMIT	The specified LED blinks once each time a message is sent from serial port 1.
COM1_RCV	The specified LED blinks once each time a message is received at serial port 1.
COM2_XMIT	The specified LED blinks once each time a message is sent from serial port 2.
COM2_RCV	The specified LED blinks once each time a message is received at serial port 2.
BKP_XMIT	The specified LED blinks once each time a backplane message is sent.
BKP_RCV	The specified LED blinks once each time a backplane message is received.

To permit a PCM task to use the **Set_led** function to control the specified LED, the number of the target task must be placed in the most significant byte of *led_definition*. The number must be in the range of valid PCM task IDs, but the specified task does not need to be active. The calling task's number, returned by **Get_task_id**, or a different task number may be specified.

A single call to **Define_led** can perform both functions. The *led_definition* may contain a task number as well as event definitions for the specified task. See the examples, below.

■ Return Value

Define_led returns a value from this table.

Return Value	_VTOS_error Value	Description
SUCCESS	Undefined	The function completed successfully.
FAILURE	BAD_ARG	The *led_number* or event definitions in *led_definition* is out of range.
	NO_TASK	The task number in *led _definition* is out of range.

■ See Also

Set_led

■ Example

```
#
include <vtos.h>

word mytask, result;

mytask = Get_task_id ();
result = Define_led ( 1, COM1_XMIT | COM1_RCV | COM2_XMIT | COM2_RCV );
result = Define_led ( 2, BKP_XMIT | BKP_RCV | (mytask << 8) );
```

This example defines LED 1 to flash whenever a message is transmitted or received on either PCM serial port. LED 2 is defined to flash whenever a message is transmitted or received on the PLC backplane, and is also configured to permit the calling task to program it with **Set_led**. Note that LED 2 will flash once per second when the PCM time-of-day clock resynchronizes itself with the PLC CPU.

Devctl_dev

Usage

```
#include <vtos.h>

word Devctl_dev ( device_name, devctl_code, data_addr, count,
                  notify_code, task_id [, <nowait options>] );
char far* device_name;
word      devctl_code;
void far* data_addr;
word      count;
word      notify_code;
word      task_id;
```

where <*nowait options*> depend on the value of *notify_code*:

```
word Devctl_dev ( device_name, devctl_code, data_addr, count,
                  WAIT, task_id );

word Devctl_dev ( device_name, devctl_code, data_addr, count,
                  EVENT_NOTIFY, task_id, local_ef_mask,
                  (device_result far*)result_ptr );
word               local_ef_mask;
device_result far* result_ptr;

word Devctl_dev ( device_name, devctl_code, data_addr, count,
                  AST_NOTIFY, task_id, ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

This function performs device control operations which are concerned with the device itself rather than a particular channel. The *device_name* must contain a valid PCM device name, ending with a colon, such as "RAM:" or "COM1:". The *task_id* is a task number returned by **Get_task_id**. The *devctl_code* specifies the device operation to be performed, and the usage of *data_addr* and *count* depend on *devctl_code*. Valid *devctl_code* values are shown in this table.

devctl_code **Value**	**Supported Devices**	**Operation**	**Description**
1	ROM:	Format device	Format the device specified by *device_name*. All data objects (files, etc.) maintained by the device will be destroyed. Currently, only the ROM: device, optionally available on the PCM 301 (GE Fanuc catalog No. IC693PCM301) supports this operation. The *data_addr* and *count* parameters are ignored.

devctl_code **Value**	**Supported Devices**	**Operation**	**Description**
2	COM1: COM2: RAM: ROM: PC:	Destroy object.	Destroy (delete) a data object maintained by the device. The *data_addr* must point to a NUL-terminated ASCII string containing the name of the object to be destroyed; *count* is ignored.
5	RAM: ROM: PC:	Get object name.	Return the name at the position specified by *count* in the list of data objects on the device specified by *device_name*. The name is copied to a NUL-terminated string at the address specified by *data_addr*; the C programmer must ensure that the memory buffer at *data_addr* is large enough for the string. If *count* exceeds the number of objects in the device's list, a NUL string is returned.
6	ROM:	Get space remaining.	Return the number of free bytes on the device. The function return contains the value, expressed as an unsigned integer. The *data_addr* and *count* values are ignored.
	COM1: COM2:	Set BREAK AST.	Specify an asynchronous trap (AST) handler function which will be called when a BREAK condition is detected on the specified device. The function address is specified in *data_addr*; and *count* is ignored. Only one task at a time may receive BREAK ASTs for each port. The task specified in *task_id* supercedes any task which may previously have set a BREAK AST for the same port.
7	COM1: COM2:	Reset BREAK AST.	Disable AST notification of BREAKs. The task specified in *task_id* must previously have called **Devctl_dev** to set BREAK AST notification for the specified device.
8	COM1: COM2:	Set ALL SENT event flag.	Specify a local event flag which will be set when all the data sent by a **Write_dev** operation to the specified device has been transmitted. The *count* parameter contains a local event flag mask specifying the event flag or flags to be set; *data_addr* is ignored. Only one task at a time may be notified of the ALL SENT condition for each port. The task specified in *task_id* supercedes any task which may previously have set an ALL SENT event flag for the same port.
9	COM1: COM2:	Reset ALL SENT event flag.	Disable ALL SENT event flag notification. The task specified in *task_id* must previously have called **Devctl_dev** to set ALL SENT notification for the specified device.

Devctl_code Value	Supported Devices	Operation	Description
10	COM1: COM2:	Mask received data errors.	Mask any combination of parity, overrun and framing errors on the specified serial port. Masking these errors prevents **Read_dev** from terminating with an error status when they occur. The `data_addr` parameter is ignored, and the `count` parameter contains a set of bits to specify the errors that will be masked: 0x0010 – Parity error mask 0x0020 – Overrun error mask 0x0040 – Framing error mask For example, 0x0010 masks parity errors only, 0x0050 masks both parity and framing errors, and 0x0070 masks all three.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Devctl_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a far pointer because its type is not specified by the function prototype in **vtos.h**. If the call succeeds, the **ioreturn** member of the structure at *result_ptr* contains SUCCESS and the **iostatus** member is undefined; when a failure occurs, **ioreturn** contains IO_FAILED, and **iostatus** contains an error status code. For a discussion of asynchronous I/O using event flags, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle contains* a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine*, it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**. If the call succeeds, the **arg2** member of the **ast_blk** contains SUCCESS and the **arg1** member is undefined; when a failure occurs, **arg2** contains IO_FAILED, and **arg1** contains an error status code. For a discussion of asynchronous I/O using AST functions, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

■ Return Value

In WAIT mode, the value returned by a successful **Devctl_dev** call depends on the *devctl_code*. When an error occurs, IO_FAILED is returned; a status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function is undefined and should be ignored. The actual return and status values are available in **device_result** and **ast_blk** structures, respectively.

For all modes, the return and status variables contain values from the following table.

Return Value	Status Value	Completion Status
Depends on devctl_code	SUCCESS	The operation was successful.
IO_FAILED	ABORTED	The operation was aborted before completion.
	UNSUPT BAD_ARG	The specified *devctl_code* is not supported by the specified device.
	NO_ACCESS	The operation attempted to delete a file which is in use or protected.
	NO_FILE	The operation attempted to delete a non-existent file.
	NO_MEMORY	The operation requires a temporary buffer which could not be allocated.
	NO_DEVICE	The *device_name* is not a valid device.

See Also

Ioctl_dev, Special_dev

Example

```
#include <vtos.h>

word status, task_id;
device_result evt_result;

char name [16];

task_id = Get_task_id ();
status = Devctl_dev ( "ROM:", 1, NULL, 0, WAIT, task_id );

status = Devctl_dev ( "RAM:", 2, "MYFILE.DAT", 0, EVENT_NOTIFY,
                            task_id, EF_03,
                            (device_result far* )&evt_result );

devctl_value = Devctl_dev ( "PC:", 5, name, 2, AST_NOTIFY,
                              task_id, devctl_ast_func, AST_DEVCTL );
```

This example uses WAIT, EVENT_NOTIFY, and AST_NOTIFY **Devctl_dev** requests to:

1. format its **ROM:** device;
2. delete the file **RAM:MYFILE.DAT**; and
3. find the file name of the second directory entry in the current directory on the current drive of a **PC:** device attached to the PCM file server port.

The PCM is assumed to be a PCM 301. The definitions of **devctl_ast_func** and **AST_DEVCTL** are not shown. See **Close_dev.**

Disable_asts

Usage

```
#include <vtos.h>

void Disable_asts ( void );
```

Description

This function prevents the calling task from executing asynchronous traps (ASTs) until they are re-enabled by a subsequent **Enable_asts** call. ASTs are often disabled while accessing data which is shared by mainline and AST functions in the same task.

If ASTs are already pending, they will not be serviced until ASTs are re-enabled by calling **Enable_asts**. **Disable_asts** may be called more than once (calls may be nested), as long as one call to **Enable_asts** is eventually made for each **Disable_asts** call.

Caution

After calling Disable_asts, never call Wait_ast before calling Enable_asts. A PCM lockup or other unexpected operation may result.

Return Value

None.

See Also

Enable_asts

Example

```
#include <vtos.h>

Disable_asts ();
/*
 * Access shared data.
 */
Enable_asts ();
```

Elapse

Usage

```
#include <vtos.h>

long unsigned Elapse ( continue_flag );
word continue_flag;
```

Description

This function returns the number of milliseconds since the last reset of its count. The count is reset when *continue_flag* is zero,

Return Value

The number of milliseconds is returned as a long unsigned integer.

See Also

Example

```
#include <vtos.h>

long unsigned ms_since_reset;
ms_since_reset = Elapse ( 0 );
```

This example reads the **Elapse** count and resets it as well.

Enable_asts

Usage

```
#include <vtos.h>

void Enable_asts ( void );
```

Description

This function enables the calling task to resume processing of asynchronous traps (ASTs). If ASTs have not been disabled, the call has no effect.

Return Value

None.

See Also

Disable_asts

Example

See **Disable_asts**.

establish_comm_session

Usage

```
#include <session.h>

REQSTAT establish_comm_session ( device_id, session_id );
char far* device_id;
BYTE far* session_id;
```

Description

This function performs the steps needed to establish a communication session with the PLC CPU. The *device_id* parameter must contain the address of a string which specifies a channel number for the session. The string must start with the '#' character, followed by one or two decimal digit characters containing an integer in the range 5 through 31, inclusive. The *session_id* parameter must contain the address of a variable declared as type BYTE. When the function returns successfully, the BYTE at *session_id* will contain a value which must be used to identify the current session in subsequent PLC API function calls.

Caution

The channel number specified in the *device_id* parameter must <u>not</u> be used in a call to Open_dev for the CPU:# device.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
API_NOT_INITIALIZED	REQUEST_ERROR	No previous call to **api_initialize** was made.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	The channel specified by *device_id* could not be opened..
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.

See Also

api_initialize, configure_comm_link

Example

See **api_initialize**.

establish_mixed_memory

Usage

```
#include <mxread.h>

REQSTAT establish_mixed_memory ( session_id, list_size,
                                 mixed_list_ptr, list_id_ptr );
BYTE                          session_id;
WORD                          list_size;
MIXED_MEMORY_READ_STRUC far*  mixed_list_ptr;
BYTE far*                     list_id_ptr;
```

Description

The functions **establish_mixed_memory**, **read_mixed_memory**, and **cancel_mixed_memory** are used together to read a collection of PLC memory data from the PLC CPU. The collection can contain up to 2 Kbytes of data, and can include up to 59 different memory references in Series 90-30 PLCs and 256 different references in Series 90-70 PLCs. This method is very efficient when the same collection of data is requested from the PLC many times.

A collection (or "shopping list") of PLC data memory formats is specified by calling **establish_mixed_memory**. Up to two different lists may be established at a time. The *mixed_list_ptr* parameter must contain the address of a **MIXED_MEMORY_READ_STRUC** structure, as defined in **mixtypes.h**. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *list_size* must contain the size in BYTEs of the **MIXED_MEMORY_READ_STRUC**, including all its memory formats. When the function completes successfully, the address specified in the *list_id_ptr* parameter will contain a unique identifier value for the list.

The **MIXED_MEMORY_READ_STRUC** structure may be thought of as:

```
typedef struct mem_format_rec {
    BYTE memory_type;
    WORD mem_offset;
    BYTE mem_length;
} MEM_FORMAT_STRUC;

typedef struct mixed_memory_read_rec {
    char             local_pblock_name[8];
    WORD             local_segment;
    WORD             num_mem_formats;
    MEM_FORMAT_STRUC mem_formats [num_mem_formats];
} MIXED_MEMORY_READ_STRUC;
```

However, standard C compilers complain about the variable length array declaration, **mem_formats [num_mem_formats]**. The array size could be declared large enough to contain the maximum number of memory formats. But there is no need to keep the shopping list around after it is passed to **establish_mixed_memory**. The best solution is to simulate a variable length array by allocating just enough free memory to hold one **MIXED_MEMORY_READ_STRUC** plus zero or more copies of **MEM_FORMAT_STRUC**. The total number of **MEM_FORMAT_STRUC**s, including the one in the **MIXED_MEMORY_READ_STRUC,** must equal the actual number of formats in the list. After the list has been established, the memory is freed. The example program below shows how to do this.

Caution

Any C source code which accesses members of a MEM_FORMAT_STRUC must be compiled with the /Zp command line option of the Microsoft C compiler. This option enables structure packing. To be recognized by the PLC CPU, the MEM_FORMAT_STRUC structure must be packed.

The **local_pblock_name** member must contain the name of a PLC program subblock when any of the memory formats specifies Series 90-70 %L data. It must contain the PLC program name when any of the memory formats specifies Series 90-70 %P data and none of the formats specify %L data. Each list may specify %L data from at most one subblock.

The **local_segment** member must contain zero. The **num_mem_formats** member must contain the number of memory formats in the list (that is, the number of array elements in **mem_formats**).

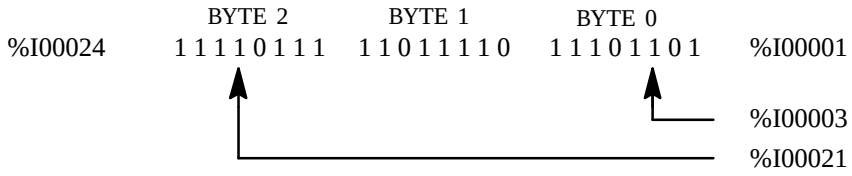
The actual "shopping list" of memory reference formats is contained in the **mem_formats** array member of the **MIXED_MEMORY_READ_STRUC** structure. Each element of **mem_formats** specifies a PLC memory reference by type, starting offset, and length.

The **memory_type** member of each **mem_formats** element must contain a valid memory type. This table shows the memory type values from **mixtypes.h** and **memtypes.h** which may be used.

Reference Type	Access Type	Data Type	*memory_type* Value
%AI	Analog Input Register Analog Input High Alarm Analog Input Low Alarm Analog Input Fault / No Fault Analog Input Diagnostic	WORD BYTE BYTE BYTE BYTE	AI_DATA AI_HIALR AI_LOALR AI_FAULT AI_DIAG
%AQ	Analog Output Register Analog Output High Alarm Analog Output Low Alarm Analog Output Fault / No Fault Analog Output Diagnostic	WORD BYTE BYTE BYTE BYTE	AQ_DATA AQ_HIALR AQ_LOALR AQ_FAULT AQ_DIAG
%R %P %L	Register Memory Program Register Memory (Series 90-70 PLC only) Local Register Memory (Series 90-70 PLC only)	WORD WORD WORD	R_DATA P_DATA L_DATA
%I	Input Status Table Input Transition Table Input Override Table Input Diagnostic Table	Discrete memory in BYTE mode.	I_STATUS_BYTE I_TRANS_BYTE I_OVRD_BYTE I_DIAG_BYTE
%Q	Output Status Table Output Transition Table Output Override Table Output Diagnostic Table	Discrete memory in BYTE mode.	Q_STATUS_BYTE Q_TRANS_BYTE Q_OVRD_BYTE Q_DIAG_BYTE
%T	Temporary Status Table Temporary Transition Table Temporary Override Table	Discrete memory in BYTE mode.	T_STATUS_BYTE T_TRANS_BYTE T_OVRD_BYTE
%M	Internal Status Table Internal Transition Table Internal Override Table	Discrete memory in BYTE mode.	M_STATUS_BYTE M_TRANS_BYTE M_OVRD_BYTE
%SA	System A Status Table System A Transition Table System A Override Table	Discrete memory in BYTE mode.	SA_STATUS_BYTE SA_TRANS_BYTE SA_OVRD_BYTE
%SB	System B Status Table System B Transition Table System B Override Table	Discrete memory in BYTE mode.	SB_STATUS_BYTE SA_TRANS_BYTE SB_OVRD_BYTE
%SC	System C Status Table System C Transition Table System C Override Table	Discrete memory in BYTE mode.	SC_STATUS_BYTE SC_TRANS_BYTE SC_OVRD_BYTE
%S	System Status Table System Transition Table System Override Table	Discrete memory in BYTE mode.	S_STATUS_BYTE S_TRANS_BYTE S_OVRD_BYTE
%G	Global Genius Status Table Global Genius Transition Table Global Genius Override Table	Discrete memory in BYTE mode.	G_STATUS_BYTE G_TRANS_BYTE G_OVRD_BYTE

Discrete Data Formats

Note that discrete memory types are specified as BYTE mode. To reduce PLC processing, discrete data is requested and returned in the format used internally by the PLC CPU. Consequently, the starting and ending references for discrete data must be specified differently than for **read_sysmem**. This figure shows the relationship of BYTE mode addressing to our usual way of thinking about discrete references.



Suppose that an application requires discrete input table values from %I00003 through %I00021, inclusive. %I00003 is in the first BYTE of the discrete input table; this BYTE is addressed as BYTE zero. When the range of interest begins with any of the inputs %I00001 through %I00008, the value in the **mem_offset** member of the corresponding **mem_formats** element must be zero. Similarly, when the range of interest starts within %I00009 through %I00016, the **mem_offset** value must be one. A simple algorithm for calculating the byte offset value for any conventional discrete reference is:

```
byte_offset = (discrete_ref – 1) / 8
```

where "/" is the C language integer division operator, which produces an integer result by simply throwing away any remainder. If you try this algorithm with any of the discrete input references from the figure above, your result should agree with the BYTE numbers above the groups of inputs. For example, %I00008 is in BYTE 0, %I00009 is in BYTE 1, and %I00024 is in BYTE 2.

The **mem_offset** of each **mem_formats** element with a discrete **memory_type** must contain a zero-based byte address calculated with this algorithm.

The BYTE mode length for discrete references is simply the BYTE offset of the ending reference minus the BYTE offset of the starting reference plus one:

```
byte_length = ending_byte_offset – starting_byte_offset + 1
```

where **ending_byte_offset** and **starting_byte_offset** are BYTE offsets calculated using the previous algorithm. For example, the BYTE length required to get two discrete references which are in the same BYTE is obviously one, which agrees with the result produced by the algorithm. Furthermore, the result using the range of references from %I00008 through %I00017, inclusive, is three; a glance at the figure shows that this is correct. The **mem_length** of each **mem_formats** element with a discrete **memory_type** must contain a BYTE length calculated with this algorithm.

Returning to the example using %I00003 through %I00021, we see that the correct **mem_offset** and **mem_length** values are zero and three, respectively. The example program below includes C source code to use these inputs in a memory list.

WORD Data Formats

The **mem_offset** values for WORD data, such as %R, are also zero-based. They are simply one less than the conventional. one-based references. For example, the offset for %R00001 is zero; the offset for %P1000 is 999.

WORD data values for **mem_length** are simply the number of words desired.

The **#define** section of the example program below contains simple methods for converting starting and ending references to **mem_offset** and **mem_length** values, for both discrete and word references.

■ Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
CONN_ID_NOT_FOUND	REQUEST_ERROR	Two mixed memory "shopping lists" already exist..
INVALID_CONN_SIZE	REQUEST_ERROR	The *list_size* specifies too many point formats. The maximum values are: 256 for Series 90-70 PLC CPUs and 59 for Series 90-30 PLC CPUs.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

■ See Also

cancel_mixed_memory, cancel_mixed_memory_nowait, establish_mixed_memory_nowait, read_localdata_nowait, read_localdata_nowait, read_mixed_memory, read_mixed_memory_nowait, read_prgmdata, read_prgmdata_nowait, read_sysmem, read_sysmem_nowait

■ Example

```
#include <mxread.h>
#include <malloc.h>
#include <string.h>
#define NUM_FMTS 2
#define FIRST_L_REF 11
#define LAST_L_REF  16   /* %L00011 through %L00016, inclusive */
#define L_SIZE      (LAST_L_REF – FIRST_L_REF + 1)
#define FIRST_I_REF  3
#define LAST_I_REF  21   /* %I00003 through %I00021, inclusive */
#define I_SIZE      ((LAST_I_REF – 1)/8 – (FIRST_I_REF – 1)/8 + 1)

main ()
{
    BYTE mixed_data [ L_SIZE * sizeof ( WORD ) + I_SIZE * sizeof ( BYTE ) ];
    WORD list_size, i_start;
    REQSTAT status;
    MIXED_MEMORY_READ_STRUC far* mmp;
    void far* dp;
    BYTE session_id;
    BYTE list_id;
/*
 * Use api_initialize, configure_comm_link, and establish_comm_session
 * to start PLC communication and initialize session_id.
 */
    list_size = MIXED_MEM_STRUC_SIZE + (NUM_FMTS – 1) * MEM_FMT_STRUC_SIZE;
    mmp = malloc( list_size );

    _fstrcpy ( mmp–>local_pblock_name, ”LOCALB” );
    mmp–>local_segment = 0;
    mmp–>num_mem_formats = NUM_FMTS;

    mmp–>mem_formats [0].memory_type = L_DATA;
    mmp–>mem_formats [0].mem_offset = FIRST_L_REF – 1;
    mmp–>mem_formats [0].mem_length = L_SIZE;

    mmp–>mem_formats [1].memory_type = I_STATUS_BYTE;
    mmp–>mem_formats [1].mem_offset = (FIRST_I_REF – 1) / 8;
    mmp–>mem_formats [1].mem_length = I_SIZE;

    status = establish_mixed_memory ( session_id, list_size, mmp, &list_id );

    if ( status != REQUEST_OK ) {
       /* investigate the error */
    } else {
       /* The new mixed memory list was established, so the  */
       /* buffer with the shopping list is no longer needed. */
       free( mmp );
       status = read_mixed_memory ( session_id, list_id, mixed_data );
       if ( status != REQUEST_OK ) {
          /* investigate the error */
       } else {
          /* The mixed memory data is available: L_SIZE WORDS of %L */
          /* data start at mixed_data, and I_SIZE BYTES of %I data  */
          /* start at mixed_data + L_SIZE * sizeof ( WORD ).        */
          /* Cancel the PLC list_id when it is no longer needed.    */
          status = cancel_mixed_memory ( session_id, list_id );

          if ( status != REQUEST_OK ) {
              /* investigate the error */
          } else {
              /* the list was cancelled */
          }
       }
    }
}
```

This program uses WAIT mode requests to establish a list for reading mixed memory, reading the data in the list, and cancelling the list.

establish_mixed_memory_nowait

Usage

```
#include <mxreadnw.h>

REQID establish_mixed_memory_nowait ( session_id, list_size,
                                      mixed_list_ptr,
                                      list_id_ptr );
BYTE                          session_id;
WORD                          list_size;
MIXED_MEMORY_READ_STRUC far*  mixed_list_ptr;
BYTE far*                     list_id_ptr;
```

Description

See **establish_mixed_memory**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
CONN_ID_NOT_FOUND	REQUEST_ERROR	Two mixed memory "shopping lists" already exist..
INVALID_CONN_SIZE	REQUEST_ERROR	The *list_size* specifies too many point formats. The maximum values are: 256 for Series 90-70 PLC CPUs and 59 for Series 90-30 PLC CPUs.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

cancel_mixed_memory, cancel_mixed_memory_nowait, establish_mixed_memory, read_localdata_nowait, read_localdata_nowait, read_mixed_memory, read_mixed_memory_nowait, read_prgmdata, read_prgmdata_nowait, read_sysmem, read_sysmem_nowait

Example

```
#include <mxread.h>
#include <malloc.h>
#include <string.h>
#define NUM_FMTS 2
#define FIRST_L_REF 11
#define LAST_L_REF  16   /* %L00011 through %L00016, inclusive */
#define L_SIZE      (LAST_L_REF – FIRST_L_REF + 1)
#define FIRST_I_REF  3
#define LAST_I_REF  21   /* %I00003 through %I00021, inclusive */
#define I_SIZE      ((LAST_I_REF – 1)/8 – (FIRST_I_REF – 1)/8 + 1)

main ()
{
    BYTE mixed_data [ L_SIZE * sizeof ( WORD ) + I_SIZE * sizeof ( BYTE ) ];
    WORD list_size, i_start;
    REQSTAT status;
    REQID request_id;
    MIXED_MEMORY_READ_STRUC far* mmp;
    void far* dp;
    BYTE session_id;
    BYTE list_id;
/*
 * Use api_initialize, configure_comm_link, and establish_comm_session
 * to start PLC communication and initialize session_id.
 */
    list_size = MIXED_MEM_STRUC_SIZE + (NUM_FMTS – 1) * MEM_FMT_STRUC_SIZE;
    mmp = malloc( list_size );

    _fstrcpy ( mmp–>local_pblock_name, ”LOCALB” );
    mmp–>local_segment = 0;
    mmp–>num_mem_formats = NUM_FMTS;

    mmp–>mem_formats [0].memory_type = L_DATA;
    mmp–>mem_formats [0].mem_offset = FIRST_L_REF – 1;
    mmp–>mem_formats [0].mem_length = L_SIZE;

    mmp–>mem_formats [1].memory_type = I_STATUS_BYTE;
    mmp–>mem_formats [1].mem_offset = (FIRST_I_REF – 1) / 8;
    mmp–>mem_formats [1].mem_length = I_SIZE;
```

```
   request_id = establish_mixed_memory_nowait ( session_id,
                                          list_size, mmp, &list_id );
   if ( request_id < REQUEST_OK ) {
      status = request_id;
   } else {
      do {
         status = reqstatus ( request_id, TRUE );
         /* do something else useful */
      } while ( status == REQUEST_IN_PROGRESS );
   }

   if ( status != REQUEST_OK ) {
      /* investigate the error */
   } else {
      /* The new mixed memory list was established, so the  */
      /* buffer with the shopping list is no longer needed. */
      free( mmp );
      request_id = read_mixed_memory_nowait ( session_id,
                                        list_id, mixed_data );
      if ( request_id < REQUEST_OK ) {
         status = request_id;
      } else {
         do {
             status = reqstatus ( request_id, TRUE );
             /* do something else useful */
         } while ( status == REQUEST_IN_PROGRESS );
      }

      if ( status != REQUEST_OK ) {
         /* investigate the error */
      } else {
         /* The mixed memory data is available: L_SIZE WORDS of %L */
         /* data start at mixed_data, and I_SIZE BYTES of %I data  */
         /* start at mixed_data + L_SIZE * sizeof ( WORD ).        */
         /* Cancel the PLC list_id when it is no longer needed.    */
         request_id = cancel_mixed_memory_nowait ( session_id, list_id );
         if ( request_id < REQUEST_OK ) {
             status = request_id;
         } else {
             do {
             status = reqstatus ( request_id, TRUE );
             /* do something else useful */
             } while ( status == REQUEST_IN_PROGRESS );
         }

         if ( status != REQUEST_OK ) {
             /* investigate the error */
         } else {
             /* the list was cancelled */
         }
      }
   }
}
```

This example uses NOWAIT mode requests to establish a list for reading mixed memory, reading the data in the list, and cancelling the list.

Get_best_buff

Usage

```
#include <vtos.h>

void far* Get_best_buff ( size_in_bytes );
long unsigned size_in_bytes;
```

Description

This function allocates PCM free memory using the ”best fit” algorithm. The *size_in_bytes* is the size of the desired memory block, which may be as large as the largest block of free memory in the PCM. When PCM free memory is fragmented, the smallest free memory block which is as large or larger than *size_in_bytes* is returned.

Return Value

When the call succeeds, **Get_best_buff** returns a **far** pointer to the newly allocated memory block. If the allocation fails, a NULL pointer is returned.

See Also

Get_buff, Max_avail_buff, Return_buff

Example

```
#include <vtos.h>

byte far* buff_ptr;
long unsigned buff_size = 66000;

if ( Max_avail_buff () < buff_size ) {
    /* sorry, can’t allocate buff_size bytes */
} else {
    buff_ptr = Get_best_buff ( buff_size );
    /* use the memory buffer */
    Return_buff ( buff_ptr );
}
```

This example allocates a 66,000 byte memory block, if one is available. The block is used and then returned to free memory.

Get_board_id

■ Usage

```
#include  <vtos.h>

board_id Get_board_id ( void );
```

■ Description

This function returns hardware identification codes for the PCM where it executes.

■ Return Value

`Get_board_id` returns a structure type, `board_id`, defined in VTOS.H. For standard Series 90-70 PCMs and Series 90-70 standalone PCMs with various daughter boards installed, the `id` member of this structure contains one of these values:

Standard Series 90-70 PCM, IC697PCM711	Series 90-70 Standalone PCM, IC697PCM712	DaughterBoard Type
0x0000	**0x0040**	No daughter board
0x001C	**0x005C**	64K memory board (192K bytes total)
0x001F	**0x005F**	128K memory board (256K bytes total)
0x001E	**0x005E**	256K memory board (384K bytes total)
0x001D	**0x005D**	512K memory board (640K bytes total)
0x001B	**0x005B**	DLAN communication board

For Series 90-70 display coprocessor modules, the `id` member of this structure contains one of these values:

Return Value	Module Type
0x0080	Series 90-70 Graphics Display Coprocessor Module with video daughter board.
0x0081	Series 90-70 Alphanumeric Display Coprocessor Module.

For Series 90-30 PCMs and derivative module types, the `id` member of this structure contains one of these values:

Return Value	Module Type
0x00FF	Series 90-30 PCM model IC693PCM300 (160K bytes).
0x00FE	Series 90-30 PCM model IC693PCM301 (192K bytes).
0x00FC	Series 90-30 PCM model IC693PCM311 (640K bytes).
0x0082	Series 90-30 Alphanumeric Display Coprocessor Module.

The `hardware_type` member of the returned structure specifies the Series 90-70 hardware type when the PCM firmware version is 4.03 or newer:

Value of `hardware_type`	Hardware Type
0	PCMA1
1	PCMA2
2	PCMA3

The `hardware_type` value is undefined in a Series 90-30 PCM and when the PCM firmware is older than version 4.03.

■ See Also

■ Example

```
#include <vtos.h>

#define 9030_PCM        0x0020
#define STANDALONE_PCM  0x0040
#define DISPLAY_COPROC  0x0080

board_id bid;

bid = Get_board_id ();
if ( bid.id & 9030_PCM ) {
    /* a Series 90-30 PCM */
} else if (bid.id & DISPLAY_COPROC) {
    /* an Alphanumeric or Graphic Display Coprocessor */
} else {
    /* a Series 90-70 PCM */
    if (Get_pcm_rev() >= 0x0403) {
      /* bid.hardware_type contains 0 (PCMA1), 1 (PCMA2) or 2 (PCMA3) */
    }
    if (bid.id & STANDALONE_PCM) {
        /* a Series 90-70 standalone PCM */
    } else {
        /* a Series 90-70 standard PCM */
    }
}
```

This example shows how to determine the PCM hardware configuration from the word value returned by `Get_board_id`.

Get_buff

Usage

```
#include <vtos.h>

void far* Get_buff ( size_in_bytes );
long unsigned size_in_bytes;
```

Description

This function allocates PCM free memory using the ”first fit” algorithm. The *size_in_bytes* is the size of the desired memory block, which may be as large as the largest block of free memory in the PCM. When PCM free memory is fragmented, the list of free blocks is searched, and the first free memory block which is as large or larger than *size_in_byte*s is returned.

Return Value

When the call succeeds, **Get_buff** returns a **far** pointer to the newly allocated memory block. If the allocation fails, a NULL pointer is returned.

See Also

Get_best_buff, Max_avail_buff, Return_buff

Example

See **Get_best_buf**.

get_config_info

Usage

```
#include <chksum.h>
#include <apitypes.h>

REQSTAT get_config_info ( session_id, config_info_ptr );
BYTE                    session_id;
CONFIG_INFO_STRUC far*  config_info_ptr;
```

Description

This function retrieves length and checksum information about the Logicmaster 90 configuration data currently stored in the PLC PCU. It can be used to determine if the configuration has changed. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *config_info_ptr* must contain the address of a structure of type **CONFIG_INFO_STRUC**, as defined in **apitypes.h**. This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure will contain data from the current PLC configuration.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_config_info_nowait, get_prgm_info, get_prgm_info_nowait

Example

```
#include <chksum.h>

REQSTAT status;
CONFIG_INFO_STRUC config_info;
status = get_config_info ( session_id, &config_info );
```

This example uses a WAIT mode request to get the length and checksums of the Logicmaster 90 configuration data in the PLC CPU.

get_config_info_nowait

Usage

```
#include <chksumnw.h>

REQID get_config_info_nowait ( session_id, config_info_ptr );
BYTE                   session_id;
CONFIG_INFO_STRUC far* config_info_ptr;
```

Description

See **get_config_info**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_config_info, get_prgm_info, get_prgm_info_nowait, reqstatus

Example

```
#include <chksumnw.h>

REQID request_id;
REQSTAT status;
CONFIG_INFO_STRUC config_info;

request_id = get_config_info_nowait ( session_id, &config_info );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the config info is available */
}
```

This example uses a NOWAIT mode request to get the length and checksums of the Logicmaster 90 configuration data in the PLC CPU.

get_cpu_type_rev

Usage

```
#include <utils.h>

REQSTAT get_cpu_type_rev ( session_id, cpu_type_rev );
BYTE                session_id;
CPU_TYPE_STRUC far* cpu_type_rev;
```

Description

This function obtains the major and minor CPU type, along with the major and minor PLC CPU software revision. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *cpu_type_rev* parameter must contain the address of a structure of type **CPU_TYPE_STRUC**, as defined in **apitypes.h**. This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure at *cpu_type_rev* will contain data from the PLC.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_cpu_type_rev_nowait

Example

```
#include <utils.h>

REQSTAT status;
CPU_TYPE_STRUC cpu_type;
status = get_cpu_type_rev ( session_id, &cpu_type );
```

This example uses a WAIT mode request to get the model number and firmware release of the PLC CPU.

get_cpu_type_rev_nowait

Usage

```
#include <utilsnw.h>

REQID get_cpu_type_rev_nowait ( session_id, cpu_type_rev );
BYTE                session_id;
CPU_TYPE_STRUC far* cpu_type_rev;
```

Description

See **get_cpu_type_rev**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_cpu_type_rev, reqstatus

Example

```
#include <utilsnw.h>

REQID request_id;
REQSTAT status;
CPU_TYPE_STRUC cpu_type;

request_id = get_cpu_type_rev_nowait ( session_id, &cpu_type );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
        /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the new PLC hardware data is available */
}
```

This example uses a NOWAIT mode request to get the model number and firmware release of the PLC CPU.

Get_date

Usage

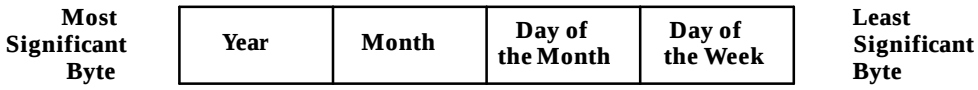
```
#include <vtos.h>

long unsigned Get_date ( void );
```

Description

This function returns a long unsigned integer which contains the current day of the week, day of the month, month and year from the PCM's local internal time of day clock, which is normally synchronized with the PLC CPU clock.

Return Value

The date format is:



See Also

`Get_time`

Example

```
#include <vtos.h>
#include <stdio.h>

byte date [ sizeof ( long ) ];

*((long *)date) = Get_date ();
printf ( "day of week = %d, day of month = %d, month = %d, year = %d\n",
    date [0], date [1], date [2], date [3] );
```

Get_dp_buff

Usage

```
#include <vtos.h>

void far* Get_dp_buff ( size_in_bytes );
word size_in_bytes;
```

Description

This function allocates a block of VMEbus dual ported memory in a Series 90-70 PCM. The *size_in_bytes* parameter specifies the memory buffer size. When the call succeeds, the buffer is reserved for exclusive use by the calling task.

Return Value

When **Get_dp_buff** completes successfully, it returns a far pointer to the allocated buffer. If a buffer of the requested size is not available or the call is made in a Series 90-30 PCM, a NULL pointer is returned and **_VTOS_error** contains NO_MEMORY.

See Also

Reserve_dp_buff, Return_dp_buff

Example

```
#include <vtos.h>

struct mystruct far* vme_memory_ptr;

vme_memory_ptr = Get_dp_buff ( 4096 );
if ( vme_memory_ptr != NULL ) {
    /* use the VMEbus memory */
    Return_dp_buff ( vme_memory_ptr );
}
```

Get_mem_lim

Usage

```
#include <vtos.h>

void far* Get_mem_lim ( void );
```

Description

This function returns the starting address of the memory block at the top of PCM memory which has been excluded from VTOS use. This address is set by the **Y** command of the PCM command interpreter. Memory in the excluded area may be used as private memory by a PCM application.

To determine the end of the private application memory area, use **Get_board_id** to find the PCM hardware type.

Return Value

After a memory limit has been set using the PCM Y command, **Get_mem_lim** returns a **far** address which is one byte above the last memory byte available to VTOS. When no limit has been set, **Get_mem_lim** returns NULL.

See Also

Get_board_id

Example

```
#include <vtos.h>

byte far* private_memory_ptr;
private_memory_ptr = Get_mem_lim ();
if ( private_memory_ptr != NULL ) {
    /* use the private memory */
}
```

get_memtype_sizes

Usage

```
#include <utils.h>

REQSTAT get_memtype_sizes ( session_id, mem_sizes );
BYTE                 session_id;
MEM_SIZES_STRUC far* mem_sizes;
```

Description

This function obtains the sizes of various PLC memory types. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *mem_sizes* parameter must contain the address of a structure of type **MEM_SIZES_STRUC**, as defined in **apitypes.h**. This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure at *mem_sizes* will contain data from the PLC.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_memtype_sizes_nowait

Example

```
#include <utils.h>

REQSTAT status;
MEM_SIZES_STRUC plc_mem_sizes;
status = get_memtype_sizes ( session_id, &plc_mem_sizes );
```

This example uses a WAIT mode request to get the sizes of the PLC memory types.

get_memtype_sizes_nowait

Usage

```
#include <utilsnw.h>

REQID get_memtype_sizes_nowait ( session_id, mem_sizes );
BYTE                 session_id;
MEM_SIZES_STRUC far* mem_sizes;
```

Description

See **get_memtype_sizes**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_memtype_sizes, reqstatus

Example

```
#include <utilsnw.h>

REQID request_id;
REQSTAT status;
MEM_SIZES_STRUC plc_mem_sizes;

request_id = get_memtype_sizes_nowait ( session_id, &plc_mem_sizes );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the memory size data is available */
}
```

This example uses a NOWAIT mode request to get the sizes of the PLC memory types.

Get_mod

Usage

```
#include <vtos.h>

mod_hdr far* Get_mod ( uppercase_module_name );
char far* uppercase_module_name;
```

Description

This function returns the address of a named memory module. The *uppercase_module_name* must point to a NUL terminated ASCII string which contains the name of a PCM memory module. The name is case sensitive, and all alphabetic characters are expected to be upper case. For more information on PCM memory modules, see chapter 5, *PCM Libraries and Header Files*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

Return Value

Get_mod returns the **far** address of the **mod_hdr** structure at the start of the named memory module. If no module with the specified name is found, NULL is returned.

See Also

Example

```
#include <vtos.h>

char name[] = "MYMOD.DAT";
mod_hdr far* module_ptr;
module_ptr = Get_mod ( name );
```

Get_next_block

Usage

```
#include <vtos.h>

arg_blk far* Get_next_block ( semaphore_ptr );
void far* semaphore_ptr;
```

Description

This function returns device argument blocks to a VTOS device driver. A future revision of this manual will discuss VTOS device drivers.

Return Value

See Also

Example

get_one_rackfaults

Usage

```
#include <faults.h>

REQSTAT get_one_rackfaults ( session_id, rack_num,
                             rack_fault_bits );
BYTE                  session_id;
BYTE                  rack_num;
RACK_FAULT_STRUC far* rack_fault_bits;
```

Description

This function permits the user to retrieve all rack, slot, genius bus, and genius device fault bits for a specified Series 90-70 PLC rack. This request is valid only for Series 90-70 PLCs. The *session_id* must be a value returned by a previous, successful call to **establish_comm_ses sion**. The *rack_num* parameter must contain the number of the desired rack; valid rack numbers are 0 through 7. The *rack_fault_bits* parameter must contain the address of a structure of type **RACK_FAULT_STRUC**, as defined in **apitypes.h**. This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure at *rack_fault_bits* will contain current PLC fault bit data.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *rack_num* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_ session**.
NO_SMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus, chk_genius_bus_nowait, chk_genius_device, chk_genius_device_nowait, get_one_rackfaults_nowait, get_rack_slot_faults, get_rack_slot_faults_nowait

Example

```
#include <faults.h>

REQSTAT status;
RACK_FAULT_STRUC rack_bits;
status = get_one_rackfaults ( session_id, rack_num, &rack_bits );
```

This example uses a WAIT mode request to determine if a fault exists in any of the PLC racks.

get_one_rackfaults_nowait

Usage

```
#include <faultsnw.h>

REQID get_one_rackfaults_nowait ( session_id, rack_num,
                                  rack_fault_bits );
BYTE                  session_id;
BYTE                  rack_num;
RACK_FAULT_STRUC far* rack_fault_bits;
```

Description

See **get_one_rackfaults**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus.** When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *rack_num* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus, chk_genius_bus_nowait, chk_genius_device, chk_genius_device_nowait, get_one_rackfaults, get_rack_slot_faults, get_rack_slot_faults_nowait, reqstatus

Example

```
#include <faultsnw.h>

REQID request_id;
REQSTAT status;
RACK_FAULT_STRUC rack_bits;

request_id = get_one_rackfaults_nowait ( session_id, rack_num, &rack_bits );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the fault data is available */
}
```

This example uses a NOWAIT mode request to determine if a fault exists in any of the PLC racks.

Get_pcm_rev

Usage

```
#include <vtos.h>

word Get_pcm_rev ( void );
```

Description

This function returns the PCM firmware revision number.

Return Value

Get_pcm_rev returns a word value with the major revision number in the high order byte and the minor revision number in the low order byte. Both are in hexadecimal format. No errors are returned.

See Also

Example

```
#include <vtos.h>
#include <stdio.h>

word pcm_rev;

pcm_rev = Get_pcm_rev ();
printf ( "The PCM firmware revision is %x.%02x.\n",
    pcm_rev >> 8, pcm_rev & 0xff );
```

This example formats the firmware revision number of the PCM where it executes and prints it to **STDOUT**. When run in a release 3.00 PCM, it prints:

```
The PCM firmware revision is 3.00.
```

get_prgm_info

■ Usage

```
#include <chksum.h>

REQSTAT get_prgm_info ( session_id, prog_info_ptr );
BYTE                    session_id;
PROGRAM_INFO_STRUC far* prog_info_ptr;
```

■ Description

This function retrieves program name, size, and checksum information about the program currently stored in the PLC CPU. It may be used to determine the program name or whether the program has changed. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *prog_info_ptr* parameter must contain the address of a structure of type **PROGRAM_INFO_STRUC**, as defined in **apitypes.h**. This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure at *prog_info_ptr* will contain data from the PLC.

■ Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_config_info, get_config_info_nowait, get_prgm_info_nowait

Example

```
#include <chksum.h>

PROGRAM_INFO_STRUC prog_info;
REQSTAT status;
status = get_prgm_info ( session_id, &prog_info );
```

This example uses a WAIT mode request to get the program name and checksums of the PLC program.

get_prgm_info_nowait

Usage

```
#include <chksumnw.h>

REQID get_prgm_info_nowait ( session_id, prog_info_ptr );
BYTE                    session_id;
PROGRAM_INFO_STRUC far* prog_info_ptr;
```

Description

See **get_prgm_info**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

get_config_info, get_config_info_nowait, get_prgm_info, reqstatus

Example

```
#include <chksumnw.h>

REQID request_id;
REQSTAT status;
PROGRAM_INFO_STRUC prog_info;

request_id = get_prgm_info_nowait ( session_id, &prog_info );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC program data is available */
}
```

This example uses a NOWAIT mode request to get the program name and checksums of the PLC program.

get_rack_slot_faults

■ Usage

```
#include <faults.h>

REQSTAT get_rack_slot_faults ( session_id, rack_num,
                               rack_slot_fault_bits );
BYTE      session_id;
BYTE      rack_num
WORD far* rack_slot_fault_bits;
```

■ Description

This function permits the user to determine if there are one or more faults on a specified Series 90-70 PLC rack (in *rack_num*), and, if so, which slot or slots within that rack contain faulted modules. This request is valid only for Series 90-70 PLCs. Valid rack numbers are 0 through 7. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. When the request has completed successfully, the WORD variable whose address is specified in *rack_slot_fault_bits* will contain the following bit pattern:

Bit	Value
Bit 0 (Least Significant Bit)	1 – There is a fault anywhere on the rack. 0 – There are no faults on the rack.
Bit 1	1 – The rack has failed (for example, been powered off). 0 – The rack is operating.
Bit 2	1 – Slot 0 in the rack is faulted. 0 – Slot 0 is not faulted.
Bit 3	1 – Slot 1 in the rack is faulted. 0 – Slot 1 is not faulted.
. . .	. . .
Bit 11	1 – Slot 9 in the rack is faulted. 0 – Slot 9 is not faulted.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus, chk_genius_bus_nowait, chk_genius_device, chk_genius_device_nowait, get_one_rackfaults, get_one_rackfaults_nowait, get_rack_slot_faults_nowait

Example

```
#include <faults.h>

WORD rack_slot_bits;
REQSTAT status;
status = get_rack_slot_faults ( session_id, rack_num, &rack_slot_bits );
```

This example uses a WAIT mode request to determine if a fault exists in any slot of the specified PLC rack.

get_rack_slot_faults_nowait

Usage

```
#include <faultsnw.h>

REQID get_rack_slot_faults_nowait ( session_id, rack_num,
                                    rack_slot_fault_bits );
BYTE      session_id;
BYTE      rack_num;
WORD far* rack_slot_fault_bits;
```

Description

See **get_rack_slot_faults**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

chk_genius_bus, chk_genius_bus_nowait, chk_genius_device, chk_genius_device_nowait, get_one_rackfaults, get_one_rackfaults_nowait, get_rack_slot_faults, reqstatus

Example

```
#include <faultsnw.h>

REQID request_id;
REQSTAT status;
WORD rack_slot_bits;

request_id = get_rack_slot_faults_nowait ( session_id, rack_num,
                                           &rack_slot_bits );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the fault data is available */
}
```

This example uses a NOWAIT mode request to determine if a fault exists in any slot of the specified PLC rack.

Get_task_id

Usage

```
#include <vtos.h>

word Get_task_id ( void );
```

Description

This function returns the task identification number (task ID) of the calling task.

Return Value

`Get_task_id` returns a 16-bit unsigned integer in the range from zero to fifteen. There are no errors.

See Also

Example

```
#include <vtos.h>

word task_id;
task_id = Get_task_id ();
```

Get_time

Usage

```
#include <vtos.h>

long unsigned Get_time ( format );
word format;
```

Description

This function returns the current time of the PCM internal time-of-day clock. A format value of zero (0) specifies that the time should be returned as a count of milliseconds since midnight. Any non-zero format value specifies an hours / minutes / seconds / hundredths format.

The PCM clock is normally synchronized to the PLC CPU time of day clock within a plus-or-minus one second tolerance.

Return Value

Get_time returns a long unsigned integer containing the current PCM time of day. When *format* is zero, the return value contains the number of milliseconds since 12:00 Midnight. The count is reset automatically to zero (0) every day at Midnight. This format is useful when calculating the time between two events. However, the **Elapse** function is better because its count is unaffected at midnight or by resynchronization of the PCM clock to the PLC.

When *format* is non-zero, the time is returned in a four byte, hours / minutes / seconds / hundredths format. The hours value is in the most significant byte, minutes in the next most significant byte, seconds in the next most significant byte, and hundredths of seconds in the least significant byte. The hour value between Midnight and 1:00 a.m. is zero, and the hour value between 1:00 p.m. and 2:00 p.m. is 13. The resolution of this format is 10 milliseconds (one hundredth second). This format is useful when the time of day is to be displayed.

See Also

Elapse, Get_date

Example

```
#include <vtos.h>
#include <stdio.h>

void main ()
{
    byte hmsh [sizeof ( long )];
    long counts = Get_time (0);
    *((long* )hmsh) = Get_time (1);
    printf ( "milliseconds since midnight = %lu\n", counts );
    printf (
       "hours/minutes/seconds/hundredths = %02d:%02d:%02d.%02d\n",
       hmsh [3], hmsh [2], hmsh [1], hmsh [0]
       );
}
```

This example gets both time formats and prints them to **STDOUT**. The byte array, **hmsh,** is defined to be the same size as a long integer. A type cast is used to assign the value returned by **Get_time** directly to it.

Init_task

Usage

```
#include <vtos.h>

void Init_task ( task_id, stack_ptr, code_ptr, data_seg, env_ptr );
word         task_id;
byte  far*   stack_ptr;
void (far*   code_ptr)();
word         data_seg;
env_blk far* env_ptr;
```

Description

This function is used primarily to execute a VTOS device driver as a task. A future revision of this manual will discuss VTOS device drivers.

Return Value

See Also

Example

Install_dev

Usage

```
#include <vtos.h>

void Install_dev ( dcb_ptr );
dcb_blk far* dcb_ptr;
```

Description

This function installs a PCM device driver. A future revision of this manual will discuss VTOS device drivers.

Return Value

None.

See Also

Example

Install_isr

Usage

```
#include <vtos.h>

void Install_isr ( interrupt_number, isr_procedure );
word       interrupt_number;
word (far* isr_procedure)();
```

Description

This function installs the address of *isr_procedure* as the interrupt vector for *interrupt_number*. It may be used to provide a software interrupt interface to functions called by more than one task.

Caution

VTOS makes extensive use of interrupts. User installed interrupt handlers should be restricted to *interrupt_number* values in the range 10 hexadecimal through 3F hexadecimal, inclusive.

The only special requirements for *isr_procedure* are that it must be declared **far** and return a word value. It does not need to preserve the registers which are overwritten by ordinary C functions.

Caution

Interrupt service routines installed by Install_isr must not be compiled using the interrupt function keyword or perform a return from interrupt on exit.

When a non-zero value is returned by *isr_procedure*, the VTOS scheduler runs immediately. This mechanism permits high priority tasks to be made ready by interrupts and start executing with the minimum time lag.

Return Value

None.

See Also

Example

```
#include <vtos.h>

word far isr_proc ( void )
{
    /* do something interesting */
    return (0)
}

void main ()
{
    Install_isr ( 0x21, isr_proc );
}
```

This example installs **isr_proc** as the interrupt service routine for software interrupt 21 hexadecimal.

Ioctl_dev

Usage

```
#include <vtos.h>

word Ioctl_dev ( device_handle, ioctl_code, notify_code,
                 task_id [, <nowait options>] );
word device_handle;
word ioctl_code;
word notify_code;
word task_id;
```

where **<***nowait options***>** depend on the value of *notify_code*:

```
word Ioctl_dev ( device_handle, ioctl_code, WAIT, task_id );

word Ioctl_dev ( device_handle, ioctl_code, EVENT_NOTIFY,
                 task_id, local_ef_mask,
                 (device_result far*)result_ptr );
word               local_ef_mask;
device_result far* result_ptr;

word Ioctl_dev ( device_handle, ioctl_code, AST_NOTIFY,
                 task_id, ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

This function performs I/O control operations on the channel specified by *device_handle*. The type of operation is specified by *ioctl_code*, which must contain a value from this table.

ioctl_code **Value**	**Supported Devices**	**Operation**	**Return Value**
1	COM1: COM2: CPU: RAM: ROM: REMn: PC: NULL:	Is *device_handle* a physical device (rather than a file)?	0 (No) -1 (Yes)
2	COM1: COM2: CPU: RAM: ROM: REMn: PC: NULL:	Are any received characters available on the specified channel?	0 (No) -1 (Yes)
3	COM1: COM2: CPU: REMn:	Is the channel ready to send output?	0 (No) -1 (Yes)
4	COM1: COM2: REMn:	Purge the channel's type-ahead buffer.	0 (Always)
5	COM1: COM2:	Turn on Break.	0 (Always)
6	COM1: COM2:	Turn off Break.	0 (Always)
7	COM1: COM2:	Has a Break been detected since the last check?	0 (No) 1 (Yes)
8	COM1: COM2: RAM: REMn: PC; NULL:	Is there a CTRL-C character in the type-ahead buffer?	0 (No) 1 (Yes)
9	COM1: COM2:	Turn on the Request To Send (RTS) output.	0 (Always)
10	COM1: COM2:	Turn off the Request To Send (RTS) output.	0 (Always)
11	COM1: COM2:	Turn on the Data Terminal Ready (DTR) output, enabling the RS-485 line drivers.	0 (Always)
12	COM1: COM2:	Turn off the Data Terminal Ready (DTR) output, disabling the RS-485 line drivers.	0 (Always)
13	COM1: COM2: REMn:	Return the number of characters in the type-ahead buffer for the specified channel.	The character count
14	REMn:	Flush the output buffer for the specified channel.	0 (Always)

The *notify_code* specifies the method used to notify the calling task that the operation has completed; its value may be WAIT, EVENT_NOTIFY, or AST_NOTIFY. When WAIT is used, there are no *nowait options*, and the return from **Ioctl_dev** is delayed until the operation completes. The other *notify_code* values cause the function to return immediately, allowing the calling task to continue execution. **Ioctl_dev** does not wait for external events, so WAIT mode is almost always used. The EVENT_NOTIFY and AST_NOTIFY forms are included for completeness.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Close_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a **far** pointer because its type is not specified by the function prototype in **vtos.h**. If the call succeeds, the **ioreturn** member of the structure at *result_ptr* contains SUCCESS and the **iostatus** member is undefined; when a failure occurs, **ioreturn** contains IO_FAILED, and **iostatus** contains an error status code. For a discussion of asynchronous I/O using event flags, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle* contains a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine*, it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**. If the call succeeds, the **arg2** member of the **ast_blk** contains SUCCESS and the **arg1** member is undefined; when a failure occurs, **arg2** contains IO_FAILED, and **arg1** contains an error status code. For a discussion of asynchronous I/O using AST functions, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

■ Return Value

In WAIT mode, the value returned by a successful **Ioctl_dev** call depends on the *ioctl_code.* When an error occurs, IO_FAILED is returned; a status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function is undefined and should be ignored. The actual return and status values are available in **device_result** and **ast_blk** structures, respectively.

For all modes, the return and status variables contain values from this table.

Return Value	Status Value	Completion Status
Depends on *ioctl_code*	SUCCESS	The operation was successful.
IO_FAILED	ABORTED	The operation was aborted before completion.
	UNSUPT	The specified *devctl_code* is not supported by the specified device.
	BAD_ARG	The specified *devctl_code* is not supported by the specified device, or *device_handle* is invalid.

See Also

Devctl_dev, Special_dev

Example

```
#include <vtos.h>

word handle, task_id;
device_result evt_result;

task_id = Get_task_id ();
handle = Open_dev ( "COM1:", READ_MODE, WAIT, task_id );

if ( Ioctl_dev ( handle, 8, WAIT, task_id ) ) {
    Reset_ef ( EF_01 );
    Ioctl_dev ( handle, 4, EVENT_NOTIFY, task_id,
                EF_01, (device_result far* )&evt_result );
}
```

This example opens serial port one for reading, and then calls **Ioctl_dev** to look for a CTRL-C character (ASCII End of Text, code 3) in the type-ahead buffer. If one was detected, **Ioctl_dev** is called again to flush the buffer.

Iset_ef

Usage

```
#include <vtos.h>

word Iset_ef ( local_ef_mask, task_id );
word local_ef_mask;
word task_id;
```

Description

This function sets one or more local event flags, specified by bits in *local_ef_mask*, for the task specified by *task_id*. The caller may specify its own task or a different one. If any of the specified event flags have already been set, they remain set. Event flags which are not specified remain unchanged. If the specified task was waiting for one of the local event flags specified in *local_ef_mask*, it is made ready.

Unlike **Set_ef**, this function does not call the VTOS scheduler directly. When **Iset_ef** is called from a communication timer timeout function or an interrupt service routine (installed by **Install_isr**), control is returned to the caller, which can continue its processing.

Return Value

Iset_ef returns one (1) if the task specified by *task_id* was made ready when the specified event flag or flags were set; otherwise it returns zero. The calling interrupt service routine should return this same value, so that the VTOS scheduler will run whenever a task is made ready.

See Also

Iset_gef, Set_ef, Set_gef, Wait_ef

Example

```
#include <vtos.h>

word task_ready;
task_ready = Iset_ef ( EF_13 | EF_6, 7 );
```

This example sets the local event flags in zero-based bits 13 and six (6) for task seven (7).

Iset_gef

Usage

```
#include <vtos.h>

word Iset_gef ( global_ef_mask );
word global_ef_mask;
```

Description

This function sets the global event flag or flags specified by *global_ef_mask*. Global event flags which were already set are unchanged. If one or more tasks are waiting for the specified global event flag or flags, they are made ready.

Unlike **Set_gef**, this function does not call the VTOS scheduler directly. When **Iset_gef** is called from a communication timer timeout function or an interrupt service routine (installed by **Install_isr**), control is returned to the caller, which can continue its processing.

Return Value

Iset_gef returns one (1) if any tasks were made ready when the event flag or flags were set; otherwise it returns zero. The calling interrupt service routine should return this same value, so that the VTOS scheduler will run whenever a task is made ready.

See Also

Iset_ef, Set_ef, Set_gef, Wait_gef

Example

```
#include <vtos.h>

word task_ready;
task_ready = Iset_gef ( EF_10 | EF_03 | EF_01 );
```

This example sets the global event flags in zero-based bits ten (10), three (3), and one (1).

Link_sem

Usage

```
#include <vtos.h>

word Link_sem ( sem_name );
char far* sem_name;
```

Description

Before a task may use a semaphore, it must call **Link_sem** to get a handle for it. The semaphore is specified by *sem_name,* which must point to a NUL terminated ASCII string. The string may contain up to seven characters plus the NUL character. Semaphore names are case sensitive: **"MY_SEM"** and **"My_Sem"** are different semaphores.

If the semaphore does not already exist, it is created. If, however, the semaphore does exist, calling **Link_sem** is equivalent to calling **Block_sem**. Consequently, a call to **Unblock_sem** must always be made after the call to **Link_sem** and before any other VTOS or PLC API function call.

Return Value

Link_sem returns a handle which is used to identify the semaphore for all subsequent operations.

See Also

Block_sem, Unink_sem, Unblock_sem

Example

```
#include <vtos.h>

char name[] = "MY_SEM";
word handle;
handle = Link_sem ( name );
/* access the resource controlled by MY_SEM */
Unblock_sem ( handle );
```

Max_avail_buff

Usage

```
#include <vtos.h>

long unsigned Max_avail_buff ( void );
```

Description

This function returns the size of the largest available free memory buffer. This value is the maximum size that can be allocated with a single call to **Get_buff** or **Get_best_buff**.

Return Value

The buffer size in bytes is returned in a long unsigned integer. The call always succeeds; there are no errors.

See Also

Max_avail_mem

Example

```
#include <vtos.h>

long unsigned largest_memory_block;
largest_memory_block = Max_avail_buff ();
```

Max_avail_mem

Usage

```
#include <vtos.h>

unsigned long Max_avail_mem ( void );
```

Description

This function returns the total size of all free memory buffers.

Return Value

The total free memory size in bytes is returned in a long unsigned integer. The call always succeeds; there are no errors.

See Also

Max_avail_buff

Example

```
#include <vtos.h>

long unsigned total_memory_available;
total_memory_available = Max_avail_mem ();
```

Notify_task

Usage

```
#include <vtos.h>

void Notify_task ( arg_block_ptr );
arg_blk far* arg_block_ptr;
```

Description

This function is used by VTOS device drivers to notify other tasks when external events occur. A future revision of this manual will discuss VTOS device drivers.

Return Value

See Also

Example

Open_dev

Usage

```
#include <vtos.h>

word Open_dev ( dev_name, open_mode, notify_code,
                task_id [, <nowait options>] );
char far* dev_name;
word      open_mode;
word      notify_code;
word      task_id;
```

where **<***nowait options***>** depend on the value of *notify_code*:

```
word Open_dev ( dev_name, open_mode, WAIT, task_id );

word Open_dev ( dev_name, open_mode, EVENT_NOTIFY, task_id,
                local_ef_mask,
               (device_result far*)result_ptr );
word              local_ef_mask;
device_result far* result_ptr;

word Open_dev ( dev_name, open_mode, AST_NOTIFY, task_id,
                ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

This function opens an input / output (I / O) channel for use by the application. The *dev_name* may be the name of a physical device (such as the PCM serial ports, **COM1:** and **COM2:**). Device names must be terminated with a colon. Optionally, the physical device and colon may be followed by the name of a data object (such as a file or PLC memory reference) which is maintained by the device. No ASCII space characters are permitted in *dev_name.* All valid PCM devices, along with data object formats (if any) for each, are listed in the following sections.

PCM Serial Ports: Either or both of the two PCM serial ports may be opened as I/O channels. No data objects are supported. The *dev_name* format for serial ports is:

```
"COM1:"
"COM2:"
```

If an optional ASCII character code is specified, **Read_dev** operations on the channel will terminate when the character is encountered in the input stream. For example, character input from a terminal can be read as lines by specifying the ASCII Carriage Return character (ASCII code 13 decimal) as the optional termination character.

```
"COM1:13"
```

An optional timeout value in the range of 0 to 4095 milliseconds can also be specified. Note that it is not possible to specify both a termination character and a value. But see the example code at **Start_com_timer**.

```
"COM1:Tnnnn"
```

PCM Remote Devices: Two (2) PLC backplane communication channels may be opened as remote devices. No data objects are supported. The *dev_name* format for remote devices is:

```
"REM1:"
"REM2:"
```

The CPU Device: Backplane communication between the PCM and PLC CPU takes place through channels opened on the CPU device. A data object name is required, as described in the following paragraphs.

PLC Data: A channel may be opened on the CPU device to access PLC data. The portion of *dev_name* after the colon (':') character specifies the starting location of the data. Data specifications use the familiar Logicmaster 90 software notation:

```
"CPU:<PLC reference>[,<qualifiers>]"
```

References consist of the '%' (per cent) character, one alphabetic character specifying the reference table type, and a numeric starting offset within the specified table. All PLC reference tables begin at offset value one (1), and leading '0' characters in the offset are ignored. Valid table types are:

Valid Table Types	Description
%I	Discrete input contacts.
%Q	Discrete output coils.
%M	Discrete internal contacts.
%T	Discrete temporary contacts..
%R	Register table.
%G	Global Genius contacts.
%AI	Analog inputs.
%AQ	Analog outputs.
%SA	Special contacts.
%SB	Special contacts.
%S	Special contacts.

For example:

```
"CPU:%I10"
"CPU:%R99"
"CPU:%AI64"
"CPU:%T024"
```

The PLC discrete data types (%I, %Q, %M, %T, %G, %SA, %SB, %SC, and %S) receive special handling. When two or more consecutive discrete references are read or written, the point data is packed in bytes.

If the specified starting offset is not on an even byte boundary (offset values 1, 9, 17, 25, ...), data points are shifted within bytes to place the starting reference on a byte boundary. When data is read from the PLC CPU, **Read_dev** returns the point at the starting reference in the least significant bit of the least significant byte, and all other points are shifted accordingly. If the device was opened in NATIVE_MODE and the number of points read is not an exact multiple of eight (8), there will be one or more points in the most significant byte which lie above the specified read size. These extra points are set to zero in the data returned by **Read_dev**, regardless of their PLC CPU values. When data is written to the PLC CPU, **Write_dev** interprets the least significant bit of the least significant byte of data from the caller as the point at the starting reference. If the device was opened in NATIVE_MODE and the number of points written is not an exact multiple of eight (8), there will be one or more points in the most significant byte which lie above the specified write size. These extra points are ignored by **Write_dev**, regardless of their values in the caller's data.

The optional <*qualifiers*> may be used to specify the transition, override, or diagnostic tables (when applicable) for the specified PLC reference. Qualifiers are valid only for discrete data.

Qualifier	Description
T	Transition table.
O	Override table.
F	Diagnostic (fault) table.

For example:

```
"CPU:%I101,O"
"CPU:%Q00100,F"
"CPU:%M1,T"
```

An optional size qualifier specifies whether the PLC bit or byte mode discrete tables will be accessed:

Qualifier	Description
1	Bit mode.
8	Byte mode.

In bit mode, the numeric offset part of the reference address is interpreted as a bit offset; in byte mode, it is interpreted as a byte offset. For example, the starting references

```
"CPU:%Q17,1"
"CPU:%Q3,8"
```

refer to the same discrete output. In byte mode, exact multiples of eight points are always read and written, and the data always begins on a byte boundary.

Bit mode is the default; if no size qualifier is specified, the device is opened in bit mode.

Table and size qualifiers may be combined. The size qualifier must be the final one:

```
"CPU:%M33,T,8"
```

Note that %GA through %GE references are not directly supported. Instead, they may be accessed as subtypes of %G, as shown in this table.

Subtype	Bit Mode Start Reference	Byte Mode Start Reference
%GA	%G1281	%G161
%GB	%G2561	%G321
%GC	%G3841	%G481
%GD	%G5121	%G641
%GE	%G6401	%G801

PLC Status: Two PLC status data types, **cpu_short_status**, and **cpu_long_status,** may be opened in READ_MODE only. They contain information about the PLC CPU, its control program, and its state, and are defined in the header file **cpu_data.h**.

```
"CPU:#SSTAT"
"CPU:#LSTAT"
```

PLC Time-of-Day: The PLC time-of-day clock may be opened in WRITE_MODE and then read or written. The data type transferred during read and write operations is **cpu_tod_rec**, defined in **cpu_data.h**. Writing to a channel opened on this device sets both the PLC and PCM time-of-day clocks to the time and date specified in the caller's data. This method sets both clocks at once. It is recommended because it avoids discrepancies between the two clocks.

```
"CPU:#TOD"
```

PLC Generic Message Channel: A PLC generic message channel may be opened to send or receive PLC backplane messages. The messages may be PLC COMMREQ messages, PLC service request messages, or messages from another PCM or other Series 90 smart module. The *dev_name* format for a generic message channel is:

```
"CPU:#<number>"
```

where **<number>** is a decimal value in the range from 5 to 120, inclusive. For example:

```
"CPU:#16"
```

PCM RAM Disk: Files on the PCM RAM disk device are opened by specifying the RAM: device plus a file name. The *dev_name* format for RAM files is:

```
"RAM:<filename>"
```

File names consist of one (1) to 13 ASCII printing characters; the space character is not permitted in file names, but all other printing characters are allowed. Lower case alphabetic characters are converted to upper case. There is no built-in notion of file extensions; file names may contain any number of dot ('.') characters in any location. Subdirectories are not supported.

For example:

```
"RAM:MYFILE.DAT"
"RAM:this.is.valid"
```

EEROM Device: The PCM 301 (GE Fanuc catalog no. IC693PCM301) provides for an optional Electrically Erasable Read Only Memory (EEROM) device. The *dev_name* format for ROM files is identical to the format for RAM files:

For example:

```
"ROM:MYFILE.DAT"
```

PC: Device: The **PC:** device supports file access on a Personal Computer (PC) from PCM applications. The PC must be attached to a PCM serial port and must be running a compatible file transfer program. At present, this capability is implemented only on DOS-based PCs. The *dev_name* format for the PC device is:

```
"PC:[<drive>:][<path>]<filename>"
```

The filename is subject to all restrictions of the PC file system and is **not** converted to upper case. An optional <*drive*> letter and <*path*> specification may be used.

For example:

```
"PC:MYFILE.DAT"
"PC:A:MYFILE.DAT"
"PC:C:\MYPATH\MYFILE.DAT"
```

The *open_mode* specifies how calling application may access the device, as shown in this table.

Mode	Description
READ_MODE	This mode permits read-only access to the device or file. If the specified file does not exist, an error occurs.
WRITE_MODE	When *dev_name* specifies a file, this mode permits read and write access. If the specified file exists, its contents are deleted; if not, a new file is created. When *dev_name* specifies a serial port or remote device, this mode permits write-only access to it.
APPEND_MODE	This mode permits read and write access to an existing file. If the specified file does not exist, an empty file is created. The file pointer is always positioned at the start of the file.
AUTO_REWIND_MODE	This mode causes the device to be rewound at the end of each transfer. Every read or write operation occurs at the start of the device or file. This mode is available only for PLC data.
NATIVE_MODE	This mode changes the unit of data size from bytes to the native size of the specified object. In all subsequent **Read_dev**, **Write_dev**, **Seek_dev**, **Ioctl_dev**, and **Special_dev** calls for this device, the specified length will be interpreted as bits, bytes, or words as appropriate for the particular data. For example, if **"CPU:%I1"** is opened in NATIVE_MODE, a read length of 8 will refer to 8 bits, rather than 8 bytes. Note, however, that opening **"CPU:%I1,8"** in NATIVE_MODE results in bytes as the unit of data size.

The *notify_code* specifies the method used to notify the calling task that the operation has completed; its value may be WAIT, EVENT_NOTIFY, or AST_NOTIFY. When WAIT is used, there are no *nowait options*, and the return from **Open_dev** is delayed until the operation completes. The other *notify_code* values cause the function to return immediately, allowing the calling task to continue execution.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Open_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a **far** pointer because its type is not specified by the function prototype in **vtos.h**. For a discussion of asynchronous I/O using event flags, see chapter 6, *PCM Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle* contains a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine*, it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**. For a discussion of asynchronous I/O using AST functions, see chapter 6, *PCM Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

Return Value

Open_dev returns an unsigned integer handle which is used to refer to the named data object in all subsequent device operations. If an error occurs, the call returns IO_FAILED.

In WAIT mode, the handle is returned by the call if there are no errors. When an error occurs, IO_FAILED is returned; a status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function call is undefined and may be ignored. Separate return and status values are available in **device_result** and **ast_blk** structures, respectively.

For all modes, the return and status variables contain values from this table.

	notify_code		
	WAIT	**EVENT_NOTIFY**	**AST_NOTIFY**
Result Structure Type:	None	**device_result**	**ast_blk**
Successful call: Device handle is in the SUCCESS is in	Function return value **_VTOS_error**	**ioreturn** member **iostatus** member	**arg2** member **arg1** member
Error detected: **IO_FAILED** is in the Error code is in	Function return value **_VTOS_error**	**ioreturn** member **iostatus** member	**arg2** member **arg1** member

See Also

Close_dev, Read_dev, Seek_dev, Write_dev

■ Example

```
#include <vtos.h>
#include <dos.h>

word handle, open_ast_error, open_ast_done;

void far open_ast_func ( ast_blk far* p )
{
    if ( p->handle == AST_OPEN ) {
        open_ast_done = 1;
        open_ast_error = p->arg1;
        if ( open_ast_error == SUCCESS ) {
            /* The operation succeeded. */
            handle = p->arg2;
        }
    }
}

void main ()
{
    char name[] = "COM1:";
    word handle, task_id;
    device_result evt_result;

    task_id = Get_task_id ();
    handle = Open_dev ( name, READ_MODE, WAIT, task_id );
    if ( _VTOS_error != SUCCESS ) {
    /* There was a problem. */
    }
    Reset_ef ( EF_01 );
    Open_dev ( name, READ_MODE, EVENT_NOTIFY,
               task_id, EF_01, (device_result far* )&evt_result );
    Wait_ef ( EF_01 );

    if ( evt_result.iostatus != SUCCESS ) {
       /* There was a problem. */
    } else {
       handle = evt_result.ioreturn;
    }

    open_ast_done = 0;
    Open_dev ( name, READ_MODE, AST_NOTIFY,
               task_id, open_ast_func, AST_OPEN );

    _disable();
    if ( !open_ast_done ) {
        Wait_ast ();
    }
    _enable();

    if ( open_ast_error != SUCCESS ) {
        /* There was a problem */
    }
}
```

This example uses WAIT, EVENT_NOTIFY, and AST_NOTIFY **Open_dev** requests to open **COM1:** in read only mode.

Post_ast

Usage

```
#include <vtos.h>

void Post_ast ( task_id, ast_routine [, <ast_data>] );
word       task_id;
void (far* ast_routine)( ast_blk far* );
```

where the optional *ast_data* consists of zero to five words:

```
void Post_ast ( task_id, ast_routine[, ast_handle[,
                ast_arg1[, ast_arg2[, ast_arg3[,
                ast_arg4]]]]] );
word ast_handle, ast_arg1, ast_arg2, ast_arg3, ast_arg4;
```

Description

This function posts an asynchronous trap (AST) to the task specified by *task_id*. A task may post an AST to itself or to a different task. If the specified task is waiting for ASTs, it will be made ready and then execute the specified *ast_routine* when its priority is sufficiently high. When a task posts an AST to itself, *ast_routine* executes before **Post_ast** returns.

Up to five words of *ast_data* may be passed to **Post_ast**, which creates a temporary **ast_blk** structure and copies the data words to its corresponding members. The **ast_blk** address is passed to *ast_routine* when it is called. The **ast_blk** is deallocated immediately when *ast_routine* returns. One or more words of *ast_data* may be omitted from the right end of the parameter list. However, **Post_ast** always copies five words at the top of its stack frame to the **ast_blk**. Any words which are not specified in the call will be undefined in the **ast_blk**.

Return Value

None.

See Also

Disable_asts, Enable_asts, Wait_ast

■ Example

```
#include <vtos.h>
#include ”myapp.h”

struct mod_data far* mod_data_p;

mod_hdr far* mod_p;

void main ()
{
    Wait_gef ( MODULE_EXISTS_GEF );
    mod_p = Get_mod ( APP_MODULE );
    mod_data_p = (struct mod_data far* )(mod_p + 1);
/*
 * When something interesting happens, send data to the other task by
 * posting an AST.
 */
    Post_ast ( mod_data_p->task_num, mod_data_p->func_ptr,
       EVENT_TAG, WORD1, WORD2, WORD3, WORD4 );
}
```

where myapp.h contains:

```
#define MODULE_EXISTS_GEF              EF_00
#define APP_MODULE                   ”APPMOD1”
#define EVENT_TAG                      EVENT1
#define WORD1                     EVENT1_DATA1
#define WORD2                     EVENT1_DATA2
#define WORD3                     EVENT1_DATA3
#define WORD4                     EVENT1_DATA4

struct mod_data {
    word task_num;
    void (far* func_ptr) ();
};
```

This example uses **Post_ast** to send a message to another task when an event occurs. The other task's *task_id* and *ast_routine* address are made known through a shared memory module, **”APPMOD1”**, whose name is defined in the common header file, **myapp.h**. This information must be stored to the shared module by the other task during its own initialization. The example code waits for the MODULE_EXISTS_GEF global event flag to be set, indicating that the shared module data is available.

When the AST is posted, the actual message is in the structure members of an **ast_blk**; they are passed to **Post_ast** as separate parameters. For this simplified example, constants are used as message data. A real application would send more interesting data.

Process_env

Usage

```
#include <vtos.h>

word Process_env ( env_ptr, bad_module_ptr);
env_blk far*   env_ptr;
char far* far* bad_module_ptr;
```

Description

This function is used to execute the task which is described by the **env_blk** structure pointed to by *env_ptr*. The *bad_module_ptr* parameter contains the address of a pointer to **char**.

Return Value

Process_env returns an unsigned integer which contains one of these values. When certain errors occur, the pointer at *bad_module_ptr* will point to NUL terminated character string containing the name of a PCM memory module, as shown in this table.

Return Value	*bad_module_ptr* Points To	Completion Status
SUCCESS	Undefined.	The call completed successfully.
MODULE_NOT_FOUND	The executable module name passed to **Process_env**.	The specified module was not found.
	A module name in the command line string for the executable module.	
ILLEGAL_MODULE_TYPE	The module name passed to **Process_env**.	The specified module is not a code module.
MODULE_IN_USE	The module name passed to **Process_env**.	The specified module is a large model code module which is already in use by another task.
TASK_ID_IN_USE	The module name passed to **Process_env**.	The task ID specified in the environment block is already in use by another task.
INSUFFICIENT_MEMORY	Undefined.	There is not enough free memory to execute the specified module.

See Also

Init_task

Example

```
#include <vtos.h>

WORD error_code;
env_blk task_env;
char far* far* bad_module_ptr;
error_code = Process_env(&task_env, &bad_module_ptr);
```

read_date

Usage

```
#include <time.h>

REQSTAT read_date ( session_id, plc_date );
BYTE                session_id;
DATE_LONG_STRUC far* plc_date;
```

Description

This function returns the internal date from the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *plc_date* parameter must contain the address of a structure of type **DATE_LONG_STRUC**, as defined in **apitypes.h**. This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure will contain the current PLC date and day of the week.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date_nowait, read_time, read_timedate, read_timedate_nowait, read_time_nowait, set_date, set_date_nowait, set_time, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <time.h>

DATE_LONG_STRUC plc_date;
REQSTAT status;
status = read_date ( session_id, &plc_date );
```

This example uses a WAIT mode request to read the internal date value in the PLC CPU.

read_date_nowait

Usage

```
#include <timenw.h>

REQID read_date_nowait ( session_id, plc_date );
BYTE                   session_id;
DATE_LONG_STRUC far*   plc_date;
```

Description

See **read_date**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_time, read_timedate, read_timedate_nowait, read_time_nowait, set_date, set_date_nowait, set_time, set_timedate, set_timedate_nowait, set_time_nowait, reqstatus

Example

```
#include <timenw.h>

REQID request_id;
REQSTAT status;
DATE_LONG_STRUC plc_date;

request_id = read_date_nowait ( session_id, &plc_date );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the date is available */
}
```

This example uses a NOWAIT mode request to read the internal date value in the PLC CPU.

Read_dev

Usage

```
#include <vtos.h>

word Read_dev ( device_handle, buffer, size, notify_code,
                task_id [, <nowait options>] );
word      device_handle;
void far* buffer;
word      size;
word      notify_code;
word      task_id;
```

where <*nowait options*> depend on the value of *notify_code*:

```
word Read_dev ( device_handle, buffer, size, WAIT, task_id );
```

```
word Read_dev ( device_handle, buffer, size, EVENT_NOTIFY,
                task_id, local_ef_mask,
               (device_result far*)result_ptr );
word              local_ef_mask;
device_result far* result_ptr;
```

```
word Read_dev ( device_handle, buffer, size, AST_NOTIFY,
                task_id, ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

This function reads an I/O channel which was previously opened; the *device_handle* must be a value returned by **Open_dev**. The *buffer* parameter contains the **far** address of a memory buffer where the data will be stored, and *size* contains the number of data items to read. If the channel was opened in NATIVE_MODE, *size* specifies a number of bits, bytes, or words, depending on the type of the requested data. Otherwise, *size* specifies a number of bytes. The caller is responsible for allocating *buffer* and ensuring that it is large enough to contain the requested data.

The *notify_code* specifies the method used to notify the calling task that the operation has completed; its value may be WAIT, EVENT_NOTIFY, or AST_NOTIFY. When WAIT is used, there are no *nowait options*, and the return from **Read_dev** is delayed until the operation completes. The other *notify_code* values cause the function to return immediately, allowing the calling task to continue execution.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Read_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a **far** pointer because its type is not specified by the function prototype in **vtos.h**. If the call succeeds, the **ioreturn** member of the structure at *result_ptr* contains the number of characters read, and the **iostatus** member contains SUCCESS; when a failure occurs, **ioreturn** contains the number of characters that had been read when the failure occurred, and **iostatus** contains an error status code. For a discussion of asynchronous I/O using event flags, see chapter 6, *PCM Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle* contains a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine,* it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**. If the call succeeds, the **arg2** member of the **ast_blk** contains the number of characters read, and the **arg1** member contains SUCCESS; when a failure occurs, **arg2** contains IO_FAILED, and **arg1** contains an error status code. For a discussion of asynchronous I/O using AST functions, see chapter 6, *PCM Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

■ Return Value

In WAIT mode, the function return value contains the number of data items actually read. When an error occurs, the return value will be less than *size*. A status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function call is undefined and may be ignored. Separate return and status values are available in **device_result** and **ast_blk structures**, respectively.

For all modes, the return and status variables contain values from this table.

Return Value	Status Value	Completion Status
Equal to *size.*	SUCCESS	The specified number of data items was read.
Less than *size.*	PAST_EOF	The end of data stream was encountered before the read was completed.
	ABORTED	An EVENT_NOTIFY or AST_NOTIFY call was aborted before the read was completed.
	BAD_HANDLE	An invalid *device_handle* was specified. No data was read.

See Also

Close_dev, Open_dev, Seek_dev, Write_dev

Example

```
#include <vtos.h>

word chars_read, task, handle, tmr_hndl, flags;
device_result result;
char buf[1024];

task = Get_task_id();
handle = Open_dev( “COM1:13”, READ_MODE | WRITE_MODE, WAIT, task);

tmr_hndl = Start_timer( RELATIVE_TIMEOUT | TASK_SPECIFIED | 7,
                        MS_COUNT_MODE, 0, 5000, EF_00 );

chars_read = Read_dev( handle, buf, sizeof( buf ),
                       EVENT_NOTIFY, task, EF_01, &result );

Wait_ef( EF_00 | EF_01 );

flags = Test_ef();

if (flags & EF_01 && result.iostatus == SUCCESS) {
    /* The number of characters is in chars_read. */
} else if (flags & EF_00) {
    /* A timeout occurred. */
} else {
    /* A device error occurred. */
}
```

This example reads lines of text, terminated by the carriage return character (ASCII code 13 decimal) from serial port 1.

read_io_fault_tbl

Usage

```
#include <faults.h>

REQSTAT read_io_fault_tbl ( session_id, io_faults_ptr );
BYTE                    session_id;
IO_FAULT_TBL_STRUC far* io_faults_ptr;
```

Description

This function returns the entire contents of the PLC I/O Fault Table. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *io_faults_ptr* parameter must contain the name of an array of structure type **IO_FAULT_TBL_STRUC**, as defined in **apitypes.h**. This array must be allocated by the caller and contain enough elements to hold the entire table. The caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the array will contain all the current entries from the I/O Fault Table.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl, clr_io_fault_tbl_nowait, clr_plc_fault_tbl, clr_plc_fault_tbl_nowait, read_io_fault_tbl_nowait, read_plc_fault_tbl, read_plc_fault_tbl_nowait

Example

```
#include <faults.h>

IO_FAULT_TBL_STRUC io_fault_tbl;
REQSTAT status;
status = read_io_fault_tbl ( session_id, &io_fault_tbl );
```

This example uses a WAIT mode request to read the I/O fault table in the PLC CPU.

read_io_fault_tbl_nowait

Usage

```
#include <faultsnw.h>

REQID read_io_fault_tbl_nowait ( session_id, io_faults_ptr );
BYTE                    session_id;
IO_FAULT_TBL_STRUC far* io_faults_ptr );
```

Description

See **read_io_fault_tbl**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl, clr_io_fault_tbl_nowait, clr_plc_fault_tbl, clr_plc_fault_tbl_nowait, read_io_fault_tbl, read_plc_fault_tbl, read_plc_fault_tbl_nowait, reqstatus

Example

```
#include <faultsnw.h>

REQID request_id;
REQSTAT status;
IO_FAULT_TBL_STRUC io_faults[ ];

request_id = read_io_fault_tbl_nowait ( session_id io_faults );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the fault data is available */
}
```

This example uses a NOWAIT mode request to read the I/O fault table in the PLC CPU.

read_localdata

Usage

```
#include <prgmem.h>

REQSTAT read_localdata ( session_id, program_task_name,
                         subblock_name, begin_addr, end_addr,
                         data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
char far* subblock_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

This function returns the specified range of %L (local) data from the specified Series 90-70 subblock in the specified main program. This request is valid only for Series 90-70 PLCs. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *program_task_name* pointer must contain the address of a NUL terminated ASCII string holding the name of the control program task that owns the target subblock, and *subblock_name* must point to a NUL terminated ASCII string holding the subblock name. Valid names consist of seven characters or less, not counting the NUL character. The *begin_addr* parameter contains the one-based word index where the target data begins, and *end_addr* contains the one-based word index where the data ends. When the function succeeds, the requested data is copied to the region of memory starting at the address in *data_buffer_ptr*. This memory buffer must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOT_FOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC program task.
INVALID_PARAMETER	REQUEST_ERROR	The *subblock_name* is not the name of a subblock in the specified program, or *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata_nowait, read_prgmdata, read_prgmdata_nowait, read_sysmem, read_sysmem_nowait, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

Example

```
#include <prgmem.h>

WORD buf[7];
REQSTAT status;

/*
 * To request %L1 through %L7, inclusive from the subblock named
 * "MYBLOCK" in the program named "MYPROG":
 */

status = read_localdata ( session_id, "MYPROG", "MYBLOCK", 1, 7, buf );

/*
 * To request %L28 only from the subblock named "SUB1" in the
 * program named "LOADER":
 */

status = read_localdata ( session_id, "LOADER", "SUB1", 28, 28, buf );
```

This example uses a WAIT mode request to read the specified ranges of %L data in the specified PLC program subblocks.

read_localdata_nowait

■ Usage

```
#include <prgmemnw.h>

REQID read_localdata_nowait ( session_id, program_task_name,
                              subblock_name, begin_addr,
                              end_addr, data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
char far* subblock_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

■ Description

See **read_localdata**.

■ Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOT_FOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC program task.
INVALID_PARAMETER	REQUEST_ERROR	The *subblock_name* is not the name of a subblock in the specified program, or *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_prgmdata, read_prgmdata_nowait, read_sysmem, read_sysmem_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

reqstatus

Example

```
#include <prgmemnw.h>

WORD buf;

REQID request_id;
REQSTAT status;

request_id = read_localdata_nowait ( session_id, "LOADER", "SUB1", 28, 28,
                                     buf );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the %L data is available */
}
```

This example uses a NOWAIT mode request to read %L28 only from subblock **"SUB1"** in the program **"LOADER"**.

read_mixed_memor y

Usage

```
#include <mxread.h>

REQSTAT read_mixed_memory ( session_id, list_id, data_ptr );
BYTE      session_id;
BYTE      list_id;
void far* data_ptr;
```

Description

This function is used to read all the PLC memory references specified in the mixed memory specification list referred to by *list_id*. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**, and *list_id* must be a value returned by a successful call to **establish_mixed_memory** or **establish_mixed_memory_nowait**. The *data_ptr* must contain the address of a block of memory large enough to hold all the data specified by the list. When the function completes successfully, the PLC data has been copied to the memory at *data_ptr*. This memory buffer must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
CONN_ID_NOT_FOUND	REQUEST_ERROR	The *list_id* is not a value returned by a successful call to **establish_mixed_memory** or **establish_mixed_memory_nowait**, or has been cancelled by calling **cancel_mixed_memory** or **cancel_mixed _memory_nowait.**
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

cancel_mixed_memory, cancel_mixed_memory_nowait, establish_mixed_memory, establish_mixed_memory_nowait, read_mixed_memory_nowait

Example

See **establish_mixed_memory**.

read_mixed_memor y_nowait

Usage

```
#include <mxreadnw.h>

REQID read_mixed_memory_nowait ( session_id, list_id, data_ptr );
BYTE      session_id;
BYTE      list_id;
void far* data_ptr;
```

Description

See **read_mixed_memory**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
CONN_ID_NOT_FOUND	REQUEST_ERROR	The *list_id* is not a value returned by a successful call to **establish_mixed_memory** or **establish_mixed_memory_nowait**, or has been cancelled by calling **cancel_mixed_memory** or **cancel_mixed _memory_nowait.**
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

cancel_mixed_memory, cancel_mixed_memory_nowait, establish_mixed_memory, establish_mixed_memory_nowait, read_mixed_memory, reqstatus

Example

See **establish_mixed_memory_nowait**.

read_plc_fault_tbl

Usage

```
#include <faults.h>

REQSTAT read_plc_fault_tbl ( session_id, plc_faults_ptr );
BYTE                       session_id;
PLC_FAULT_TBL_STRUC far*   plc_faults_ptr;
```

Description

This function returns the entire contents of the PLC Fault Table. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *plc_faults_ptr* parameter must contain the name of an array of structure type **PLC_FAULT_TBL_STRUC**, as defined in **apitypes.h**. This array must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the entire table. After a successful return, the array will contain all the current entries from the PLC Fault Table.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl, clr_io_fault_tbl_nowait, clr_plc_fault_tbl, clr_plc_fault_tbl_nowait, read_io_fault_tbl, read_io_fault_tbl_nowait, read_plc_fault_tbl_nowait

Example

```
#include <faults.h>

PLC_FAULT_TBL_STRUC plc_faults[16];
REQSTAT status;
status = read_plc_fault_tbl ( session_id, plc_faults );
```

This example uses a WAIT mode request to read the PLC fault table in the PLC CPU.

read_plc_fault_tbl_nowait

Usage

```
#include <faultsnw.h>

REQID read_plc_fault_tbl_nowait ( session_id, plc_faults_ptr );
BYTE                    session_id;
PLC_FAULT_TBL_STRUC far* plc_faults_ptr );
```

Description

See **read_plc_fault_tbl**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

clr_io_fault_tbl, clr_io_fault_tbl_nowait, clr_plc_fault_tbl, clr_plc_fault_tbl_nowait, read_io_fault_tbl, read_io_fault_tbl_nowait, read_plc_fault_tbl, reqstatus

Example

```
#include <faultsnw.h>

REQID request_id;
REQSTAT status;
PLC_FAULT_TBL_STRUC plc_faults[16];

request_id = read_plc_fault_tbl_nowait ( session_id, plc_faults );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the fault data is available */
}
```

This example uses a NOWAIT mode request to read the PLC fault table in the PLC CPU.

read_prgmdata

Usage

```
#include <prgmem.h>

REQSTAT read_prgmdata ( session_id, program_task_name,
                        begin_addr, end_addr, data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

This function returns the specified range of %P (program) data from the specified Series 90-70 program. This request is valid only for Series 90-70 PLCs. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *program_task_name* pointer must contain the address of a NUL terminated ASCII string holding the name of the target program. Valid names consist of seven characters or less, not counting the NUL character. The *begin_addr* parameter contains the one-based word index where the target data begins, and *end_addr* contains the one-based word index where the data ends. When the function succeeds, the requested data is copied to the region of memory starting at the address in *data_buffer_ptr*. This memory buffer must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOTFOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC program task.
INVALID_PARAMETER	REQUEST_ERROR	The *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_prgmdata_nowait, read_sysmem, read_sysmem_nowait, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

Example

```
#include <prgmem.h>

REQSTAT status;
WORD data_buffer[96-12+1];
status = read_prgmdata ( session_id, "MYPROG", 12, 96, data_buffer );
```

This example uses a WAIT mode request to read %P12 through %P96, inclusive, from a PLC program called **"MYPROG"**.

read_prgmdata_nowait

Usage

```
#include <prgmemnw.h>

REQID read_prgmdata_nowait ( session_id, program_task_name,
                             begin_addr, end_addr,
                             data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

See **read_prgmdata**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOTFOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC programtask.
INVALID_PARAMETER	REQUEST_ERROR	The *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_prgmdata, read_sysmem, read_sysmem_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

Example

```
#include <prgmemnw.h>
/*
 * Program Data Range                begin_addr Value   end_addr Value
 *
 *  %P1 through %P24, inclusive              1                24
 *
 * %P39 through %P43, inclusive             39                43
 */

WORD data_buffer[24–1+1];

REQID request_id;
REQSTAT status;

request_id = read_prgmdata_nowait ( session_id, ”LOADER”, 1, 24, data_buffer );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the %P data is available */
}
```

This example uses a NOWAIT mode request to read the specified ranges of %P data in the PLC.

read_sysmem

Usage

```
#include <sysmem.h>

REQSTAT read_sysmem ( session_id, memory_type, begin_addr,
                      end_addr, data_buffer_ptr );
BYTE      session_id;
BYTE      memory_type;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

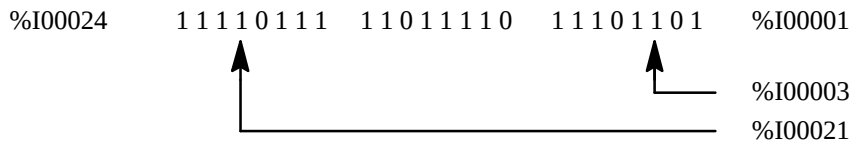
Description

This function returns the specified range of data from the specified PLC data type (%I, %Q, %R, etc.) in the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *memory_type* parameter specifies the target memory type, *begin_addr* contains the index where the target data begins, and *end_addr* contains the index where the data ends. The *end_addr* value must be greater than or equal to *begin_addr*, and must not be larger than the size returned by **get_memtype_sizes** for the specified *memory_type*. The *begin_addr* value must be a value in the range from one (1) to *end_addr*, inclusive.

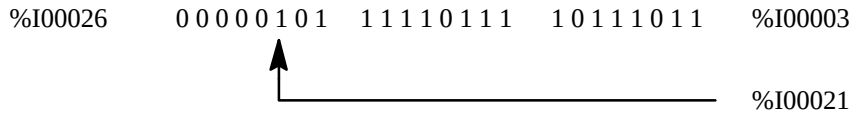
When the function succeeds, the requested data is copied to the region of memory starting at the address in *data_buffer_ptr*. This memory buffer must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data.

PLC register data types, %R, %AI, and %AQ, are treated as 16 bit binary integer values. The actual register content, however, may be signed or unsigned integers, floating point values, or text.

The PLC discrete data types, %I, %Q, %M, %T, %S, %SA, %SB, %SC, and %G, are returned in one or more bytes, with the bit specified by *begin_addr* in the least significant bit of the least significant byte. For example, consider the inputs %I00003 through %I00021. The Logicmaster 90 programming software Reference Tables display shows these inputs grouped into bytes, with %I00001 through %I00008 in the rightmost byte, etc. Our target range, %I00003 through %I00021, spans portions of three bytes:



A **read_sysmem** request for %I00003 through %I00021 will return these three bytes. The data at *begin_addr* is shifted into the least significant bit of the least significant byte. Note that the bits in the final byte which are beyond the specified range are set to zero.

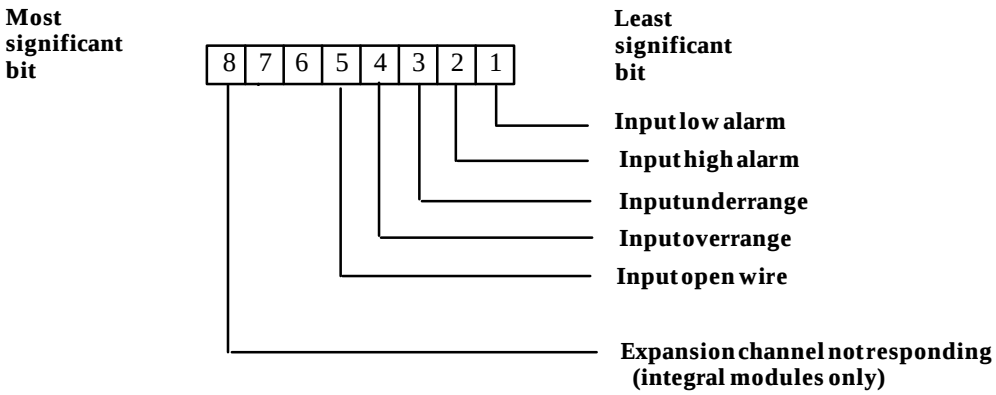


The following table shows *memory_type* values from **memtypes.h** which are valid in **read_sysmem** requests.

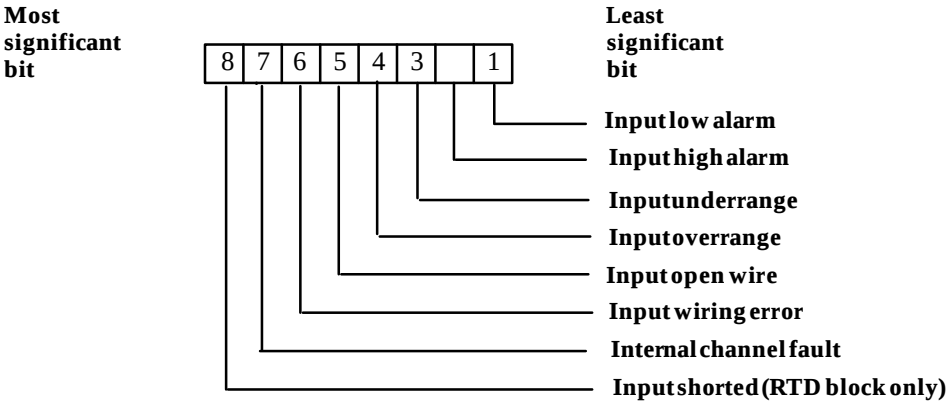
Reference Type	Access Type	Data Type	memory_type Value
%AI	Analog Input Register Analog Input High Alarm Analog Input Low Alarm Analog Input Fault / No Fault Analog Input Diagnostic	WORD BYTE BYTE BYTE BYTE	AI_DATA AI_HIALR AI_LOALR AI_FAULT AI_DIAG
%AQ	Analog Output Register Analog Output High Alarm Analog Output Low Alarm Analog Output Fault / No Fault Analog Output Diagnostic	WORD BYTE BYTE BYTE BYTE	AQ_DATA AQ_HIALR AQ_LOALR AQ_FAULT AQ_DIAG
%R	Register Memory	WORD	R_DATA
%I	Input Status Table Input Transition Table Input Override Table Input Diagnostic Table	Discrete	I_STATUS I_TRANS I_OVRD I_DIAG
%Q	Output Status Table Output Transition Table Output Override Table Output Diagnostic Table	Discrete	Q_STATUS Q_TRANS Q_OVRD Q_DIAG
%T	Temporary Status Table Temporary Transition Table Temporary Override Table	Discrete	T_STATUS T_TRANS T_OVRD
%M	Internal Status Table Internal Transition Table Internal Override Table	Discrete	M_STATUS M_TRANS M_OVRD
%SA	System A Status Table System A Transition Table System A Override Table	Discrete	SA_STATUS SA_TRANS SA_OVRD
%SB	System B Status Table System B Transition Table System B Override Table	Discrete	SB_STATUS SB_TRANS SB_OVRD
%SC	System C Status Table System C Transition Table System C Override Table	Discrete	SC_STATUS SC_TRANS SC_OVRD

Reference Type	Access Type	Data Type	memory_type Value
%S	System Status Table System Transition Table System Override Table	Discrete	S_STATUS S_TRANS S_OVRD
%G	Global Genius Status Table Global Genius Transition Table Global Genius Override Table	Discrete	G_STATUS G_TRANS G_OVRD

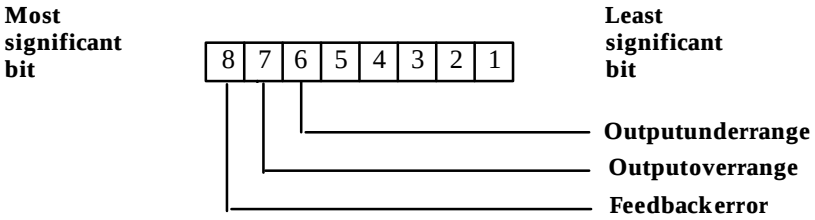
Analog input diagnostic bytes (**AI_DIAG**) for Genius analog blocks (except RTD and Thermocouple blocks) and Series 90-70 integral analog input modules contain the fault information shown in this table.



Analog input diagnostic bytes (**AI_DIAG**) for Genius RTD and Thermocouple blocks contain the fault information shown in this table.



Analog output diagnostic bytes (**AQ_DIAG**) for Genius analog blocks and Series 90-70 integral analog output modules contain the fault information shown in this table.



Analog input high alarm (**AI_HIALR**) and low alarm (**AI_LOALR**) references are BOOLEAN values which are TRUE when the corresponding **AI_DATA** value is above its high alarm limit or below its low alarm limit, respectively.

The Series 90-70 types %GA, %GB, %GC, %GD, and %GE are accessed as subtypes of %G data, as shown in this table.

Subtype	Start Reference
%GA	%G01281
%GB	%G02561
%GC	%G03841
%GD	%G05121
%GE	%G06401

For example, %GB00001 is accessed as %G02561, %GD00005 as %G06405, etc.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *end_addr* is less than *begin_addr* or out of range.
NULL_SEGSEL_PTR	REQUEST_ERROR	The *memory_type* is not supported.
INVALID_SELECTOR	REQUEST_ERROR	The *memory_type* is invalid.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem_nowait, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

■ Example

```
#include <sysmem.h>

REQSTAT status;
BYTE sesn_id;
BYTE buff[26];

/* Read discrete output status %Q00001 through %Q00024: */
status = read_sysmem ( sesn_id, Q_STATUS, 1, 24, buff );

/* Read discrete temporary transitions %T00017 through %T00208: */
status = read_sysmem ( sesn_id, T_TRANS, 17, 208, buff );

/* Read discrete internal status %M00035: */
status = read_sysmem ( sesn_id, M_STATUS, 35, 35, buff );

/* Read discrete internal overrides %M00097 through %M00112: */
status = read_sysmem ( sesn_id, M_OVRD, 97, 112, buff );

/* Read registers %R00093 through %R00098: */
status = read_sysmem ( sesn_id, R_DATA, 93, 98, buff );
```

This example uses WAIT mode requests to read various PLC data.

read_sysmem_nowait

Usage

```
#include <sysmemnw.h>

REQID read_sysmem_nowait ( session_id, memory_type, begin_addr,
                           end_addr, data_buffer_ptr );
BYTE      session_id;
BYTE      memory_type;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

See **read_sysmem**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *end_addr* is less than *begin_addr* or out of range.
NULL_SEGSEL_PTR	REQUEST_ERROR	The *memory_type* is not supported.
INVALID_SELECTOR	REQUEST_ERROR	The *memory_type* is invalid.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

Example

```
#include <sysmemnw.h>

REQID reqid1, reqid2;
REQSTAT stat1, stat2;
BYTE sesn_id;
WORD ai_data[4];
BYTE ai_diag[4];

/* Read analog input data %AI0001 through %AI0004: */
reqid1 = read_sysmem_nowait ( sesn_id, AI_DATA, 1, 4, ai_data );

if ( reqid1 < REQUEST_OK ) {
    stat1 = reqid1;
} else {
    stat1 = reqstatus ( reqid1, TRUE );
}
/* Read analog input diagnostics %AI0001 through %AI0004: */
reqid2 = read_sysmem_nowait ( sesn_id, AI_DIAG, 1, 4, ai_diag );

if ( reqid2 < REQUEST_OK ) {
    stat2 = reqid2;
} else {
    stat2 = reqstatus ( reqid2, TRUE );
}
while ( stat1 == REQUEST_IN_PROGRESS || stat2 == REQUEST_IN_PROGRESS ) {
    if ( stat1 == REQUEST_IN_PROGRESS ) {
       stat1 = reqstatus ( reqid1, TRUE );
    }
    if ( stat2 == REQUEST_IN_PROGRESS ) {
       stat2 = reqstatus ( reqid2, TRUE );
    }
}
if ( stat1 != REQUEST_OK || stat2 != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the new analog input data is available */
}
```

This example uses NOWAIT mode requests to read analog inputs and their diagnostic data.

read_time

Usage

```
#include <time.h>

REQSTAT read_time ( session_id, plc_time );
BYTE            session_id;
TIME_STRUC far* plc_time;
```

Description

This function returns the current time of day from the PLC CPU clock. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *plc_time* parameter must contain the address of a structure of type **TIME_STRUC**, as defined in **apitypes.h.** This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure will contain the current PLC time of day.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_timedate, read_timedate_nowait, read_time_nowait, set_date, set_date_nowait, set_time, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <time.h>

TIME_STRUC plc_time;
REQSTAT status;
status = read_time ( sesn_id, &plc_time );
```

This example uses a WAIT mode request to read the PLC internal time.

read_time_nowait

Usage

```
#include <timenw.h>

REQID read_time_nowait ( session_id, plc_time );
BYTE            session_id;
TIME_STRUC far* plc_time;
```

Description

See **read_time**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_timedate_nowait, reqstatus, set_date, set_date_nowait, set_time, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <timenw.h>

TIME_STRUC plc_time;
REQID request_id;
REQSTAT status;

request_id = read_time_nowait ( sesn_id, &plc_time );

if ( request_id < REQUEST_OK ) {
 status = request_id;
} else {
 do {
    status = reqstatus ( request_id, TRUE );
    /* do something else useful */
 } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC time is available */
}
```

This example uses a NOWAIT mode request to read the PLC internal time.

read_timedate

Usage

```
#include <time.h>

REQSTAT read_timedate ( session_id, plc_time_date );
BYTE                    session_id;
TIMESTAMP_LONG_STRUC far* plc_time_date;
```

Description

This function returns the current time of day, date, and day of week from the PLC CPU clock. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *plc_time* parameter must contain the address of a structure of type **TIMESTAMP_LONG_STRUC**, as defined in **apitypes.h**. This structure must be allocated by the caller; the caller is responsible for ensuring that the allocated memory is large enough to hold the requested data. After a successful return, the structure will contain the current PLC time and date.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate_nowait, read_time_nowait, set_date, set_date_nowait, set_time, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <time.h>

TIMESTAMP_LONG_STRUC plc_time_date;
REQSTAT status;
status = read_timedate ( sesn_id, &plc_time_date );
```

This example uses a WAIT mode request to read the PLC internal time and date.

read_timedate_nowait

Usage

```
#include <timenw.h>

REQID read_timedate_nowait ( session_id, plc_time_date );
BYTE                       session_id;
TIMESTAMP_LONG_STRUC far*  plc_time_date;
```

Description

See **read_timedate**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_time_nowait, reqstatus, set_date, set_date_nowait, set_time, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <timenw.h>

REQID request_id;
REQSTAT status;
TIMESTAMP_LONG_STRUC plc_time_date

request_id = read_timedate_nowait ( sesn_id, &plc_time_date );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the new time and date are available */
}
```

This example uses a NOWAIT mode request to read the PLC internal time and date.

release_request_id

Usage

```
#include <utilsnw.h>

BOOLEAN release_request_id ( request_id );
REQID request_id;
```

Description

This function is called to free *request_id* and return it to the pool of available requests. The *request_id* must be a value returned by a previous, successful nowait service request. Calling this function is necessary only when **reqstatus** is called with the **release_id** parameter FALSE. This practice is **not** recommended.

Return Value

The function returns a BOOLEAN value which is TRUE when *request_id* has been returned to the free pool, and FALSE otherwise. If the call is made before the request has completed, FALSE will be returned. The call should never be made before **reqstatus** indicates the request has completed.

See Also

reqstatus

Example

```
#include <utilsnw.h>

BOOLEAN released;
released = release_request_id ( request_id );
```

This example releases a request_id previously uses by a NOWAIT mode request.

reqstatus

Usage

```
#include <utilsnw.h>

REQSTAT reqstatus ( request_id, release_id );
REQID    request_id;
BOOLEAN  release_id;
```

Description

This function returns the current completion status of a nowait service request specified by *request_id*. If *release_id* is TRUE and the request has completed, the specified *request_id* is released. If *release_id* is FALSE or the request has not completed, *request_id* is not released. When *release_id* is FALSE and the request has completed, the application must explicitly release *request_id* at a later time. The recommended practice is to set *release_id* TRUE always.

Return Value

The function returns a REQSTAT value which contains the completion status of the NOWAIT request specified by *request_id*. Values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
0	REQUEST_IN_PROGRESS	The request is still being processed.
0	REQUEST_OK	The request completed successfully.
Minor error status code	Major error status code	The request was rejected. The major and minor error codes contain values explaining the rejection. Each NOWAIT request function reference contains the codes for that function.

All minor error codes are negative 8 bit integers. Consequently, a REQSTAT value which indicates a request was rejected may be detected by testing for a negative value.

See Also

release_request_id

Example

```
#include <utilsnw.h>

REQID request_id;
REQSTAT status;
BYTE major, minor;

request_id = start_plc_noio_nowait ( sesn_id );
status = reqstatus ( request_id, release_id );

if ( status == REQUEST_IN_PROGRESS ) {
    /* The request has not completed.                    */
} else if ( status == REQUEST_OK ) {
    /* The request completed successfully.               */
} else {
    /* There was an error.  The major and minor          */
    /* error codes contain the reason for the error.     */
    major = MAJOR_ERR ( status );
    minor = MINOR_ERR ( status );
}
```

This example calls **reqstatus** to determine the status of a previous NOWAIT mode request. Note that use of the MAJOR_ERR and MINOR_ERR macros requires that **status** must **not** be a register variable.

Reserve_dp_buff

Usage

```
#include <vtos.h>

int Reserve_dp_buff( buf_address, size_in_bytes );
void far* buf_address;
word      size_in_bytes;
```

Description

This functions reserves a memory buffer in the VMEbus dual port memory of a Series 90-70 PCM for exclusive use by the calling task. The *buf_address* and *size_in_bytes* parameters specify the location and size of the buffer, respectively.

Return Value

If the function completes successfully, SUCCESS is returned. If the specified buffer is not available or the call is made in a Series 90-30 PCM, FAILURE is returned, and the global variable **_VTOS_error** contains NO_MEMORY.

See Also

Get_dp_buff, Return_dp_buff

Example

```
#include <vtos.h>

byte far* dp_ptr = set_seg(0xA000);
ptr_off(dp_ptr) = 0x4000;
if ( Reserve_dp_buff ( dp_ptr, 4096 ) != FAILURE ) {
    /* use the buffer */
    Return_dp_buff ( dp_ptr );
}
```

Reset_ef

Usage

```
#include <vtos.h>

void Reset_ef ( local_ef_mask );
word local_ef_mask;
```

Description

This function clears one or more local event flags, specified by bits in *local_ef_mask*, for the calling task. If any of the specified event flags have already been cleared, they remain cleared. Event flags which are not specified remain unchanged. Note that local event flags can be cleared only by the task where they are local, although they can be set by other tasks. Tasks should reset any local event flags specified in a **Wait_ef** call before making the **Wait_ef** call.

Return Value

None.

See Also

Iset_ef, Iset_gef, Reset_gef, Set_ef, Set_gef, Wait_ef, Wait_gef

Example

```
#include <vtos.h>

Reset_ef ( 0xffff );
```

This example clears all the local event flags for the calling task.

Reset_gef

Usage

```
#include <vtos.h>

void Reset_gef ( global_ef_mask );
word global_ef_mask;
```

Description

This function clears one or more global event flags, specified by bits in *global_ef_mask*. If any of the specified event flags have already been cleared, they remain cleared. Event flags which are not specified remain unchanged. Tasks should reset any global event flags specified in a **Wait_gef** call before making the **Wait_gef** call.

Return Value

none.

See Also

Iset_ef, Iset_gef, Reset_ef, Set_ef, Set_gef, Wait_ef, Wait_gef

Example

```
#include <vtos.h>

Reset_gef ( EF_15 | EF_14 | EF_13 | EF_12  );
```

This example clears the four specified global event flags.

Resume_task

Usage

```
#include <vtos.h>

void Resume_task ( task_id );
word task_id;
```

Description

This function is called to resume execution of a task which was suspended by calling **Suspend_task**. The *task_id* must contain the task number of the task to be resumed.

When one task resumes a different task with higher priority, the newly resumed task will begin to execute immediately. The calling task will not return from the **Resume_task** call until the higher priority task is suspended or waits.

Return Value

None.

See Also

Suspend_task

Example

```
#include <vtos.h>

Resume_task ( 6 );
```

This example resumes execution of task six (6).

Return_buff

Usage

```
#include <vtos.h>

word Return_buff ( buffer_ptr );
void far* buffer_ptr;
```

Description

This functions returns a memory buffer to PCM free memory. The *buffer_ptr* must contain a **far** pointer to a memory buffer which was obtained by calling **Get_buff** or **Get_best_buff**.

Return Value

If the function completes successfully, SUCCESS is returned. Otherwise, FAILURE is returned, and the global variable **_VTOS_error** contains BAD_BUFFER.

During program development, it is a good idea to check for error codes from **Return_buff**. Two very common errors are to return a buffer pointer that has been changed and to return a buffer twice. These errors can corrupt the VTOS free memory list and cause symptoms with no obvious relationship to the actual error. Checking the return value from **Return_buff** is the best method for discovering these errors.

See Also

Get_buff, Get_best_buff

Example

```
#include <vtos.h>

word status;
byte far* p;

p = Get_buff ( BUFFSIZE );

if ( p != NULL ) {
     /* use the memory buffer */
    status = Return_buff ( p );
}
```

Return_dp_buff

Usage

```
#include <vtos.h>

int Return_dp_buff ( buf_address );
void far* buf_address;
```

Description

This functions returns a memory buffer in the VMEbus dual port memory of a Series 90-70 PCM. The *buf_address* parameter must contain a **far** pointer to a memory buffer which was either returned by **Get_dp_buff** or successfully reserved by **Reserve_dp_buff**.

Return Value

If the function completes successfully, SUCCESS is returned. Otherwise, FAILURE is returned, and the global variable **_VTOS_error** contains NO_MEMORY.

During program development, it is a good idea to check for error codes from **Return_dp_buff**. Two very common errors are to return a buffer pointer that has been changed and to return a buffer twice. These errors can corrupt the VMEbus dual port memory and cause symptoms with no obvious relationship to the actual error. Checking the return value from **Return_dp_buff** is the best method for discovering these errors.

See Also

Get_dp_buff, Reserve_dp_buff

Example

See **Get_dp_buff** or **Reserve_dp_buff**.

Seek_dev

Usage

```
#include <vtos.h>

word Seek_dev ( device_handle, position, notify_code,
                task_id [, <nowait options>] );
word           device_handle;
long unsigned  position;
word           notify_code;
word           task_id;
```

where **<*nowait options*>** depend on the value of ***notify_code***:

```
word Seek_dev ( device_handle, position, WAIT, task_id );
```

```
word Seek_dev ( device_handle, position, EVENT_NOTIFY,
                task_id, local_ef_mask,
                (device_result far*)result_ptr );
word              local_ef_mask;
device_result far* result_ptr;
```

```
word Seek_dev ( device_handle, position, AST_NOTIFY, task_id,
                 ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

This function positions the data pointer of the I/O channel specified by ***device_handle*** to a specified ***position*** relative to the start of the data stream. The next **Read_dev** or **Write_dev** operation will occur at the specified ***position.*** The ***device_handle*** must be a value returned by a previous, successful call to **Open_dev**. If the device was opened using NATIVE_MODE, ***position*** is interpreted in units of bits, bytes, or words, as appropriate to the device's data; otherwise, ***position*** is interpreted as bytes.

Note that there is no VTOS service to return the current position in an I/O stream. If the application needs to seek to a position other than the start of the stream, it must calculate that position.

The ***notify_code*** specifies the method used to notify the calling task that the operation has completed; its value may be WAIT, EVENT_NOTIFY, or AST_NOTIFY. When WAIT is used, there are no ***nowait options***, and the return from **Seek_dev** is delayed until the operation completes. The other ***notify_code*** values cause the function to return immediately, allowing the calling task to continue execution.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Seek_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a **far** pointer because its type is not specified by the function prototype in **vtos.h**. If the call succeeds, the **ioreturn** member of the structure at *result_ptr* contains SUCCESS and the **iostatus** member is undefined; when a failure occurs, **ioreturn** contains IO_FAILED, and **iostatus** contains an error status code. For a discussion of asynchronous I/O using event flags, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle* contains a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine*, it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**. If the call succeeds, the **arg2** member of the **ast_blk** contains SUCCESS and the **arg1** member is undefined; when a failure occurs, **arg2** contains IO_FAILED, and **arg1** contains an error status code. For a discussion of asynchronous I/O using AST functions, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

■ Return Value

In WAIT mode, SUCCESS is returned when there are no errors. When an error occurs, IO_FAILED is returned; a status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function call is undefined and may be ignored. Separate return and status values are available in **device_result** and **ast_blk** structures, respectively.

For all modes, the return and status variables contain values from this table.

Return Value	Status Value	Completion Status
SUCCESS	Undefined	The function completed successfully.
IO_FAILED	PAST_EOF	The specified *position* is past the end of the datastream.
	ABORTED	An EVENT_NOTIFY or AST_NOTIFY call was aborted before the function was completed.
	BAD_HANDLE	An invalid *device_handle* was specified.

See Also

Devctl_dev, Open_dev, Read_dev, Write_dev

Example

```
#include <vtos.h>
#include <string.h>

word chars_read, i, task, handle, seek_status;
char buf[1024];
char target[] = ”target”;

seek_status = SUCCESS + 1;
task = Get_task_id();
handle = Open_dev( ”RAM:MY.TXT”, READ_MODE | WRITE_MODE, WAIT, task );
chars_read = Read_dev( handle, buf, sizeof( buf ), WAIT, task );

if (_VTOS_error == SUCCESS) {
    for (i = 0; i < sizeof( buf ) – strlen( target ); ++i ) {
       if (strnicmp( buf+i, target, strlen( target ) ) == 0) {
           seek_status = Seek_dev ( handle, i, WAIT, task );
           break;
       }
    }
}

if (seek_status == SUCCESS) {
    /* The file is positioned at the start of the target string. */
}
```

This example calls **Seek_dev** to position the file pointer of a PCM RAM disk file to the location of a target string.

The file is opened, and the first 1024 characters are read into **buf**. Then the buffer is searched for the target string, using the case-insensitive string compare function. If the search succeeds, **Seek_dev** sets the file pointer to the location where the target was found.

Send_vme_interrupt

Usage

```
#include <vtos.h>

int Send_vme_interrupt( id );
byte id;
```

Description

`Send_vme_interrupt` may be used to generate a VMEbus interrupt from a Series 90–70 standalone PCM, IC697PCM712. There is no effect when `Send_vme_interrupt` is called from either a standard Series 90–70 PCM, 0IC697PCM711, or a Series 90–30 PCM with release 4.00 or later firmware.

Caution

If `Send_vme_interrupt` is called from any PCM with firmware earlier than release 4.00, the PCM will reset itself. The example below shows how to avoid this problem.

`Send_vme_interrupt` is provided in PCM C toolkit versions 1.04 and later. Attempting to use it in earlier versions of the toolkit will cause a linkage error (unresolved external reference).

The Series 90–70 standalone PCM can assert an interrupt request on IRQ7 only. When `Send_vme_interrupt` is called and the VMEbus interrupt handler for IRQ7 polls for an interrupt ID, the value that was passed in the *`id`* parameter will be read by the interrupt handler and used to identify the source of the interrupt. The *`id`* value must be assigned by the interrupt handler when the system is initialized. The example code for this function explains the Series 90–70 convention for obtaining *`id`*.

This function is provided for use with VMEbus masters other than Series 90–70 PLC CPUs. All versions of Series 90–70 PLC CPU firmware through release 5.xx log a fault to the PLC Fault Table when `Send_vme_interrupt` is used.

Return Value

When called from a program executing in a Series 90–70 standalone PCM, `Send_vme_interrupt` always returns SUCCESS. When called from a standard Series 90–70 PCM or Series 90–30 PCM with release 4.00 or later firmware, `Send_vme_interrupt` always returns FAILURE.

See Also

Example

```
#include <vtos.h>
#define STANDALONE_PCM  0x0040

byte far* p;
int result;

if (Get_board_id() & STANDALONE_PCM) {
    FP_SEG(p) = 0xA000;
    FP_OFF(p) = 0x006b;
    result = Send_vme_interrupt( *p );
}
```

This example verifies that it is running in a Series 90–70 standalone PCM and then calls `Send_vme_interrupt` to send an interrupt.

The *`id`* parameter value for `Send_vme_interrupt` is read from offset 6b hexadecimal (107 decimal) in the module's VME dual port memory. This location is used by Series 90–70 PLC CPUs to assign a unique interrupt vector to every smart module in the PLC. We recommend that you use the same location for this purpose in Series 90–70 standalone PCM applications.

The value of the *`id`* parameter has meaning only for the VME module that handles the IRQ7 interrupt.

set_date

Usage

```
#include <time.h>

REQSTAT set_date ( session_id, plc_date );
BYTE                session_id;
DATE_LONG_STRUC far* plc_date;
```

Description

This function allows the user to set the internal date in the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *plc_date* pointer must contain the address of a structure of type **DATE_LONG_STRUC** where the user has stored the new PLC date.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	One or more of the structure members of *plc_date* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_timedate_nowait, read_time_nowait, set_date_nowait, set_time, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <time.h>

DATE_LONG_STRUC plc_date;
REQSTAT status;
status = set_date ( sesn_id, &plc_date );
```

This example uses a WAIT mode request to set the PLC internal date.

set_date_nowait

Usage

```
#include <timenw.h>

REQID set_date_nowait ( session_id, plc_date );
BYTE                  session_id;
DATE_LONG_STRUC far*  plc_date;
```

Description

See **set_date**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	One or more of the structure members of *plc_date* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_timedate_nowait, read_time_nowait, reqstatus, set_date, set_time, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <timenw.h>

REQID request_id;
REQSTAT status;
DATE_LONG_STRUC plc_date;

request_id = set_date_nowait ( sesn_id, &plc_date );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
    /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC date was set */
}
```

This example uses a NOWAIT mode request to set the PLC internal date.

Set_dbd_ctl

Usage

```
#include <vtos.h>

void Set_dbd_ctl ( control_reg_value );
word control_reg_value;
```

Description

This function is called to set the Series 90-70 PCM 711 daughter board control register, at the PCM microprocessor I/O port address **01a0** hexadecimal. The least significant eight bits of the *control_reg_value* are stored in the task control block of the calling task. Whenever the VTOS scheduler switches between application tasks, the newly executing task's *control_reg_value* is written to the daughter board control register. **Set_dbd_ctl** is provided for controlling daughter boards other than memory expansion boards.

Return Value

None.

See Also

Set_vme_ctl

Example

Set_ef

Usage

```
#include <vtos.h>

void Set_ef ( local_ef_mask, task_id );
word local_ef_mask;
word task_id;
```

Description

This function sets one or more local event flags, specified by the bits in *local_ef_mask*, for the task specified in *task_id*. If any of the specified event flags have already been set, they remain set. Event flags which are not specified remain unchanged. If the specified task was waiting for local event flags, it is made ready.

Unlike **Iset_ef**, this function should **not** be called from an interrupt service routine or communication timer routine. When **Set_ef** readies the specified task as a result of setting event flags, it calls the VTOS scheduler. If the interrupt service routine has lower priority, control is **not** returned to the interrupt service routine, resulting in unexpected operation.

Return Value

None.

See Also

Iset_ef, Iset_gef, Reset_ef, Reset_gef, Set_gef, Wait_ef, Wait_gef

Example

```
#include <vtos.h>

Set_ef ( EF_15 | EF_14 );
```

This example sets two global event flags.

Set_gef

Usage

```
#include <vtos.h>

void Set_gef ( global_ef_mask );
word global_ef_mask;
```

Description

This function sets one or more global event flags, specified by the bits in *global_ef_mask*. If any of the specified event flags have already been set, they remain set. Event flags which are not specified remain unchanged. If one or more tasks are waiting for the specified global event flags, the highest priority waiting task is made ready.

Unlike **Iset_gef**, this function should **not** be called from an interrupt service routine. When **Set_gef** readies one or more tasks as a result of setting event flags, it calls the VTOS scheduler. If the interrupt service routine has lower priority than one of these tasks, control is **not** returned to the interrupt service routine, resulting in unexpected operation.

Return Value

None.

See Also

Iset_ef, Iset_gef, Reset_ef, Reset_gef, Set_ef, Wait_ef, Wait_gef

Example

```
#include <vtos.h>

Set_gef ( EF_03 );
```

This example sets one global event flag.

Set_led

Usage

```
#include <vtos.h>

word Set_led ( led_number, led_mode );
word led_number;
word led_mode;
```

Description

This function is called to set the state of one of two PCM light emitting diodes (LEDs). The top LED reports the operational status of the PCM and is not programmable. LED 1, the center LED, and LED 2, the bottom LED, may be programmed by **Set_led**. The *led_number* must contain one (1) or two (2), to specify LED 1 or LED 2, respectively. The *led_mode* must contain one of the values from this table.

led_mode	**Description**
LED_ON	Turn the specified LED on.
LED_OFF	Turn the specified LED off.
BLINK_LED	Blink the specified LED once.
FLASH_LED	Flash the specified LED continuously.

Before an LED state can be set by **Set_led**, the LED must be configured to permit the calling task to program it. The application can call **Define_led** to configure the LED; it can also be done in the **PCMEXEC.BAT** file which starts the application.

Only one PCM task at a time may control each LED.

Return Value

The return and **_VTOS_error** values from **Set_led** are shown in this table.

Return Value	Status Value	Completion Status
SUCCESS	Undefined	The function completed successfully.
FAILURE	BAD_ARG	The specified *led_number* or *led_mode* is invalid.
	NO_TASK	The LED specified by *led_number* is not defined for control by the calling task.

See Also

Define_led

Example

```
#include <vtos.h>

Set_led ( 1, LED_ON );
Set_led ( 2, FLASH_LED );
```

Set_local_date

Usage

```
#include <vtos.h>

typedef struct {
    byte day_of_week;
    byte day_of_month;
    byte month;
    byte year;
} ymd_date;

typedef struct {
    word lo;
    word hi;
} hilo_date;

typedef union {
    ymd_date   ymd;
    hilo_date  hilo;
    unsigned long             longdate;
} vtos_date;

int Set_local_date( hi_date, lo_date );
word hi_date;
word lo_date;
```

Description

`Set_local_date` is used to initialize the date maintained by the Series 90–70 standalone PCM, IC697PCM712. In the standard Series 90–70 PCM, IC697PCM711, and Series 90–30 PCMs, the date is automatically set to a value read from the PLC CPU. However, the standalone PCM is unable to do so, and its date is undefined until it is initialized by calling `Set_local_date`.

The new date is specified in the *`hi_date`* and *`lo_date`* parameters. Use the union type `vtos_date` to assign values to them, as shown in the example, below. Valid ranges of date values are:

Parameter	Range
Day of month	1 .. last day of specified month
Month	1 .. 12
Year	0 .. 99

Year values in the range 80 .. 99 are assumed to be 1980 through 1999, and year values zero through 79 are assumed to be 2000 through 2079. February 29 is a valid day of month for leap years, but is not valid otherwise.

`Set_local_date` calculates the correct day of week for the specified date. The calculated values range from zero through six, corresponding to Sunday through Saturday. If a day of week value is passed in *hi_date* and *lo_date*, it is ignored.

Calling `Set_local_date` has no effect in a standard Series 90–70 PCM or a Series 90–30 PCM with release 4.00 or later firmware.

Caution

If `Set_local_date` is called from any PCM with firmware earlier than release 4.00, the PCM will reset itself. The example below shows how to avoid this problem.

`Set_local_date` is provided in PCM C toolkit versions 1.04 and later. Attempting to use it in earlier versions of the toolkit will cause a linkage error (unresolved external reference).

■ Return Value

`Set_local_date` returns zero when the operation succeeds. If the specified date is invalid, BAD_ARG is returned.

In a standard Series 90–70 PCM or Series 90–30 PCM with release 4.00 or later firmware, `Set_local_date` returns the value in the *`lo_date`* parameter.

■ See Also

Get_date, Set_local_time

■ Example

```
#include <vtos.h>
#include <stdio.h>
#define STADNDALONE_PCM  0x0040

char* weekdays[] = {
        ”Sunday”, ”Monday”, ”Tuesday”, ”Wednesday”,
        ”Thursday”, ”Friday”, ”Saturday”, ”BAD DAY” };

char* months[] = {
        ”BAD MONTH”, ”January”, ”February”, ”March”,
        ”April”, ”May”, ”June”, ”July”, ”August”, ”September”,
        ”October”, ”November”, ”December”, ”BAD MONTH” };

int result;

date.ymd.day_of_month  = 29;
date.ymd.month         = 2;
date.ymd.year          = 92;

if (Get_board_id () & STANDALONE_PCM) {
    result = Set_local_date( date.hilo.hi, date.hilo.lo );

    if (!result) {
        date.longdate = Get_date();
        printf( ”date = %s, %02d %s ’%02d\n”,
                (char far* )weekdays[date.ymd.day_of_week],
                 date.ymd.day_of_month,
                (char far* )months[date.ymd.month],
                 date.ymd.year );
    }
}
```

This example verifies that it is running in a Series 90–70 standalone PCM. If so, the local date is set to 29 February 1992, a leap year day. Then `Get_date` is called, and the full date, including the day of week, is printed. The result should be:

```
date = Saturday, 29 February ’92
```

Set_local_time

■ Usage

```
#include <vtos.h>

typedef struct {
    byte hundredths;
    byte secs;
    byte mins;
    byte hours;
} hmsh_time;

typedef struct {
    word lo;
    word hi;
} hilo_time;

typedef union {
    hmsh_time  hmsh;
    hilo_time  hilo;
    unsigned long     longtime;
} vtos_time;

int Set_local_time( mode, hi_time, lo_time );
word mode;
word hi_time;
word lo_time;
```

■ Description

`Set_local_time` is used to initialize the time maintained by the Series 90–70 standalone PCM, IC697PCM712. In a standard Series 90–70 PCM, IC697PCM711, or a Series 90–30 PCM, the time is automatically synchronized with the PLC CPU time of day clock. However, the standalone PCM is unable to do so, and its time is undefined until it is initialized by calling `Set_local_time`.

The new time is specified in the *hi_time* and *lo_time* parameters, and *mode* specifies whether *hi_time* and *lo_time* should be interpreted as hours, minutes, seconds and hundredths of seconds or as a count of milliseconds since midnight. Use the union type `vtos_time` to assign time values, as shown in the example, below. Valid ranges of time values are:

Parameter	Range
mode	MS_SINCE_MIDNIT, TIME_STRUCT
Milliseconds since midnight	0 .. 86399999 (decimal)
Hours	0 .. 23
Minutes	0 .. 59
Seconds	0 .. 59
Hundredths	0 .. 99

Calling `Set_local_time` has no effect in a standard Series 90–70 PCM or a Series 90–30 PCM with release 4.00 or later firmware.

Caution

If `Set_local_time` is called from any PCM with firmware earlier than release 4.00, the PCM will reset itself. The example below shows how to avoid this problem.

`Set_local_time` is provided in PCM C toolkit versions 1.04 and later. Attempting to use it in earlier versions of the toolkit will cause a linkage error (unresolved external reference).

■ Return Value

`Set_local_time` returns zero when the operation succeeds. If the specified time is invalid, BAD_ARG is returned.

In a standard Series 90–70 PCM or Series 90–30 PCM with release 4.00 or later firmware, the return value of `Set_local_time` is undefined.

■ See Also

Get_time, Set_local_date

■ Example

```
#include <vtos.h>
#include <stdio.h>
#define STANDALONE_PCM  0x0040

vtos_time time;
int result;

time.hmsh.hundredths   = 0;
time.hmsh.secs         = 0;
time.hmsh.mins         = 0;
time.hmsh.hours        = 8;

if (Get_board_id () & STANDALONE_PCM) {
    result = Set_local_time( TIME_STRUCT,
                             time.hilo.hi, time.hilo.lo
);

    if (!result) {
        time.longtime = Get_time( MS_SINCE_MIDNIT );
        printf( "time = %08lx\n", time.longtime );

        time.longtime = Get_time( TIME_STRUCT );
        printf( "time = %02d:%02d:%02d.%02d\n",
                                time.hmsh.hours,
                                time.hmsh.mins,
                                time.hmsh.secs,
                                time.hmsh.hundredths );
    }
```

```
    result = Set_local_time( MS_SINCE_MIDNIT, 0, 0 );

    if (!result) {
        time.longtime = Get_time( MS_SINCE_MIDNIT );
        printf( "time = %08lx\n", time.longtime );

        time.longtime = Get_time( TIME_STRUCT );
        printf( "time = %02d:%02d:%02d.%02d\n",
                                time.hmsh.hours,
                                time.hmsh.mins,
                                time.hmsh.secs,
                                time.hmsh.hundredths );
    }
}
```

This example verifies that it is running in a Series 90–70 standalone PCM. If so, the local time is set to exactly 8:00 a.m. using hours / minutes / seconds / hundredths format. Then `Get_time` is called twice to return the time in both formats. Next, the time is set to midnight using milliseconds format, and two more calls to `Get_time` return it in both formats. The program should print this output:

```
time = 01b77400
time = 08:00:00.01
time = 00000000
time = 00:00:00.00
```

Set_std_device

Usage

```
#include <vtos.h>

word Set_std_device ( task_id, stdio_number, device_handle );
word task_id
word stdio_number;
word device_handle;
```

Description

This function is used to redirect one of the predefined standard I/O channels, **STDIN**, **STDOUT**, or **STDERR**, specified by *stdio_number*, for the task specified by *task_id.* The specified stream is redirected to the channel specified by *device_handle*, which must contain a value returned by a successful call to **Open_dev**. The caller may redirect standard I/O for itself or a different task.

Caution

Set_std_device does no error checking. Using a *task_id* value greater than 15 can destroy VTOS data and cause mysterious errors or PCM lockup.

Return Value

Set_std_device always returns SUCCESS.

See Also

Example

```
#include <vtos.h>

word handle, task_id, status;

handle = Open_dev ( ”RAM:MYTASK.OUT”, WRITE_MODE, WAIT, task_id );
status = Set_std_device ( task_id, STDOUT, handle );
```

This example redirects standard output from the calling task to the PCM RAM disk file ”MYTASK.OUT”.

set_time

Usage

```
#include <time.h>

REQSTAT set_time ( session_id, plc_time_date );
BYTE            session_id;
TIME_STRUC far* plc_time;
```

Description

This function allows the user to set the internal time in the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *plc_time* pointer must contain the address of a structure of type **TIME_STRUC** where the user has stored the new PLC time.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	One of more of the structure members of *plc_time* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_timedate_nowait, read_time_nowait, set_date, set_date_nowait, set_timedate, set_timedate_nowait, set_time_nowait

Example

```
#include <time.h>

TIME_STRUC plc_time
/*
 * Assign the desired time values to
 * the members of plc_time.
 */

REQSTAT status;
status = set_time ( sesn_id, &plc_time );
```

This example uses a WAIT mode request to set the PLC internal time.

set_time_nowait

Usage

```
#include <timenw.h>

REQID set_time_nowait ( session_id, plc_time );
BYTE            session_id;
TIME_STRUC far* plc_time;
```

Description

See **set_time**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus.** When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	One of more of the structure members of *plc_time* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize, configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_timedate_nowait, read_time_nowait, reqstatus, set_date, set_date_nowait, set_time, set_timedate, set_timedate_nowait

Example

```
#include <timenw.h>

REQID request_id;
REQSTAT status;
TIME_STRUC plc_time

request_id = set_time_nowait ( sesn_id, &plc_time );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC time has been set */
}
```

This example uses a NOWAIT mode request to set the PLC internal time.

set_timedate

Usage

```
#include <time.h>

REQSTAT set_timedate ( session_id, plc_time_date );
BYTE                   session_id;
TIMESTAMP_LONG_STRUC far* plc_time_date;
```

Description

This function allows the user to set the internal time and date in the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *plc_time_date* pointer must contain the address of a structure of type **TIMESTAMP_LONG_STRUC** where the user has stored the new PLC time and date.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	One of more of the structure members of *plc_time_date* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_timedate_nowait, read_time_nowait, set_date, set_date_nowait, set_time, set_timedate_nowait, set_time_nowait

Example

```
#include <time.h>

TIMESTAMP_LONG_STRUC plc_time_date;
/*
 * Assign the desired time and date
 * values to the members of plc_time_date.
 */

REQSTAT status;
status = set_timedate ( sesn_id, &plc_time_date );
```

This example uses a WAIT mode request to set the PLC internal time and date.

set_timedate_nowait

Usage

```
#include <timenw.h>

REQID set_timedate_nowait ( session_id, plc_time_date );
BYTE                       session_id;
TIMESTAMP_LONG_STRUC far*  plc_time_date;
```

Description

See **set_timedate**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	One of more of the structure members of *plc_time_date* is out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

read_date, read_date_nowait, read_time, read_timedate, read_timedate_nowait, read_time_nowait, reqstatus, set_date, set_date_nowait, set_time, set_timedate, set_time_nowait

Example

```
#include <timenw.h>

REQID request_id;
REQSTAT status;
TIMESTAMP_LONG_STRUC plc_time_date;

request_id = set_timedate_nowait ( sesn_id, &plc_time_date );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC time and date have been set */
}
```

This example uses a NOWAIT mode request to set the PLC internal time and date.

Set_vme_ctl

Usage

```
#include <vtos.h>

void Set_vme_ctl ( vme_block_num, address_modifier_code );
word vme_block_num;
word address_modifier_code;
```

Description

This function is used to set the target address range for direct VMEbus access from the Series 90-70 PCM. The Series 90-70 PCM is capable of operating as a bus master on the Series 90-70 VMEbus, and can read or write VME memory on any Series 90-70 smart module (but **not** the PLC CPU) or third party VME module. Three independent VME address spaces are accessible from the PCM: standard non-privileged, short non-privileged, and short supervisory.

The VME standard non-privileged address space covers 16 Megabytes. It is mapped into a 64 Kbyte sliding window in the PCM. This window appears at segment 0B000 hexadecimal in the PCM, and spans the address range from 0B000:0000 through 0B000:0FFFF. The entire VME address space, as seen by the PCM, comprises 256 non-overlapping 64 Kbyte segments. **Set_vme_ctl** sets the PCM window to make any one of these segments visible. The *vme_block_num* parameter specifies the VME segment to be placed in the PCM window, and *address_modifier_code* contains STD_NON_PRIV.

The 16 Megabytes of VME standard non-privileged memory are assigned to Series 90-70 rack and slot addresses as shown in the following table.

Table 1. GE Fanuc Series 90-70 Module Address Allocation for Standard Access AM Code – 39H

Rack Number	Slot Number/Address Allocation							
	2	**3**	**4**	**5**	**6**	**7**	**8**	**9**
0	000000 to 01FFFF	020000 to 03FFFF	040000 to 05FFFF	060000 to 07FFFF	080000 to 09FFFF	0A0000 to 0BFFFF	0C0000 to 0DFFFF	0E0000 to 0FFFFF
0	100000 through 7FFFFF user-defined for rack 0 only.							
1	E00000 to E1FFFF	E20000 to E3FFFF	E40000 to E5FFFF	E60000 to E7FFFF	E80000 to E9FFFF	EA0000 to EBFFFF	EC0000 to EDFFFF	EE0000 to EFFFFF
2	D00000 to D1FFFF	D20000 to D3FFFF	D40000 to D5FFFF	D60000 to D7FFFF	D80000 to D9FFFF	DA0000 to DBFFFF	DC0000 to DDFFFF	DE0000 to DFFFFF
3	C00000 to C1FFFF	C20000 to C3FFFF	C40000 to C5FFFF	C60000 to C7FFFF	C80000 to C9FFFF	CA0000 to CBFFFF	CC0000 to CDFFFF	CE0000 to CFFFFF
4	B00000 to B1FFFF	B20000 to B3FFFF	B40000 to B5FFFF	B60000 to B7FFFF	B80000 to B9FFFF	BA0000 to BBFFFF	BC0000 to BDFFFF	BE0000 to BFFFFF
5	A00000 to A1FFFF	A20000 to A3FFFF	A40000 to A5FFFF	A60000 to A7FFFF	A80000 to A9FFFF	AA0000 to ABFFFF	AC0000 to ADFFFF	AE0000 to AFFFFF
6	900000 to 91FFFF	920000 to 93FFFF	940000 to 95FFFF	960000 to 97FFFF	980000 to 99FFFF	9A0000 to 9BFFFF	9C0000 to 9DFFFF	9E0000 to 9FFFFF
7	800000 to 81FFFF	820000 to 83FFFF	840000 to 85FFFF	860000 to 87FFFF	880000 to 89FFFF	8A0000 to 8BFFFF	8C0000 to 8DFFFF	8E0000 to 8FFFFF

The VME short access address spaces (short non-privileged and short supervisory) are just 64 Kbytes wide. The entire short non-privileged or short supervisory space for each Series 90-70 rack fits the PCM window, and is mapped to PCM addresses 0B000:0000 through 0B000:0FFFF. To access either of these address spaces in any Series 90-70 rack, the *vme_block_num* value must be zero and the *address_modifier_code* contains a value from this table.

Rack	*address_modifier_code*	
	Short Non-Privileged	**Short Supervisory**
0	SHORT_NP_RACK0	SHORT_SUP_RACK0
1	SHORT_NP_RACK1	SHORT_SUP_RACK1
2	SHORT_NP_RACK2	SHORT_SUP_RACK2
3	SHORT_NP_RACK3	SHORT_SUP_RACK3
4	SHORT_NP_RACK4	SHORT_SUP_RACK4
5	SHORT_NP_RACK5	SHORT_SUP_RACK5
6	SHORT_NP_RACK6	SHORT_SUP_RACK6
7	SHORT_NP_RACK7	SHORT_SUP_RACK7

The assignments of address ranges within the short access address spaces are shown in this table. Each slot within a rack is assigned its own 2 Kbyte range.

Table 2. GE Fanuc Series 90-70 Module Address Allocation for Short Access AM Codes

Slot	Address Range (Hexadecimal)
Power Supply	None
1	None
2	1000H – 17FFH
3	1800H – 1FFFH
4	2000H – 27FFH
5	2800H – 2FFFH
6	3000H – 37FFH
7	3800H – 3FFFH
8	4000H – 47FFH
9	4800H – 4FFFH
User-defined	5000H – FFFFH

Return Value

None.

See Also

Example

```
#include <vtos.h>
#include <dos.h>

word vme_data, far* p;
long vme_addr;

FP_SEG(p) = VME_WIN_SEG;
FP_OFF(p) = 0x0100;
vme_addr = 0x820000;
Set_vme_ctl ( vme_addr/0x10000L, STD_NON_PRIV );
vme_data = *p;
```

This example sets the PCM VME window to start at VMEbus Standard Non-Privileged address 820000 hexadecimal, which is assigned to Series 90-70 rack 7, slot 3. One word of data is read from offset 100 hexadecimal in the target VME memory.

Special_dev

Usage

```
#include <vtos.h>

word Special_dev ( device_handle, special_code, data_addr,
                   count, notify_code, task_id
                   [, <nowait options>] );
word      device_handle;
word      special_code;
void far* data_addr;
word      count;
word      notify_code;
word      task_id;
```

where <*nowait options*> depend on the value of *notify_code*:

```
word Special_dev ( device_handle, special_code, data_addr,
                   count, WAIT, task_id );

word Special_dev ( device_handle, special_code, data_addr,
                   count, EVENT_NOTIFY, task_id, local_ef_mask,
                   (device_result far*)result_ptr );
word               local_ef_mask;
device_result far* result_ptr;

word Special_dev ( device_handle, special_code, data_addr,
                   count, AST_NOTIFY, task_id,
                   ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

Special_dev performs several device-specific functions, as described in this table.

special_code	Supported Devices	Operation	Result
1	RAM: ROM: PC: NULL:	Get file size.	Return the size of the file or other data object specified by *device_handle*. The size is returned as a long unsigned integer at the address specified in *data_addr*; *count* is ignored, and any value may be passed.
2	PC: NULL:	Set file size.	Set the size of the data object specified by *device_handle*. The size is specified as a long unsigned integer at the address in *data_addr*; *count* is ignored, and any value may be passed.

special_code	**Supported Devices**	**Operation**	**Result**
5	CPU:	Set password.	Specify a password to establish a new PLC access privilege level, or disable synchronization of the PCM time of day clock with the PLC CPU. The channel specified by *device_handle* must be on the CPU: device. The count value is ignored. The format of the parameter string specified by data_addr is described in CPU Setup Strings (special_code = 5).
	COM1: COM2:	Set serial communication parameters.	Set the serial port communication parameters for the PCM serial port channel specified by *device_handle*. The address of a string containing the new parameters must be in *data_addr*; and *count* is ignored. The parameter string is described on the next page.
6	CPU:	Set destination address.	Set the Series 90 rack / slot destination address for generic messages sent from the CPU: device channel specified by *device_handle*. The *data_addr* parameter should contain the address of a **msg_addr** structure, as defined in CPU_DATA.H, and *count* is ignored.
7	RAM: ROM:	Set access mode.	Set the access mode of PCM files. The access mode values are one (1), indicating read only access, and zero (0), indicating read/write access. Mode values are specified in *count; data_addr* is ignored, and any address may be passed.
8	CPU:	Set segment / offset.	Set the PLC CPU reference memory type and offset for reads and writes on the CPU: device channel specified by *device_handle*. The *data_addr* parameter should contain the address of a **spec_8_struct** structure, as defined in VTOS.H, and *count* is ignored.
9	CPU:	Set high priority.	Send the generic messages written to the CPU: device channel specified by *device_handle* as high priority messages. The *data_addr* and *count* are ignored.
10	CPU:	Set timeout.	Set a timeout for **Read_dev** and **Write_dev** transfers which occur on the channel specified by *device_handle* that was opened on the CPU: device. When a transfer times out before it completes, control returns to the task which initiated the transfer, and the return status for the operation will indicate that a timeout error occurred.
11	CPU:	Get element size.	Return the element size of the data object specified by *device_handle*. The element size is the number of bits in the unit element. Size parameters for **Read_dev**, **Write_dev**, and **Seek_dev** are always expressed as a number of unit elements, which are usually eight-bit bytes. Other elements sizes are often used for the CPU: device, which accesses discrete (single bit) and word (16-bit) data. The element size is returned as an unsigned integer at the address specified in *data_addr*; *count* is ignored, and any value may be passed.

Serial Port Setup Strings (`special_code = 5`)

Serial port (COM1:**,** COM2:**)** setup strings for `special_code = 5` have this format:

```
<baud_rate>,<parity>,<data_bits>,<stop_bits>,<flow_control>,
<physical_interface>,<duplex_mode>,<delay_value>,<typeahead_size>
```

where:

<baud_rate> = 300, 600, 1200, 2400, 4800, 9600, 19200*, or 38400 – the number of bits per second. Note that 38,400 baud is supported only by the Series 90–70 PCM, and only for RS–422 or RS–485 port configurations.

<parity> = O, E, N* – the type of parity checking: Odd, Even, or None.

<data_bits> = 7 or 8* – the number of data bits per character. Use 8 unless text with 7 bit characters will be the *only* data transferred.

<stop_bits> = 1* or 2 – the number of stop bits per character. The normal selection for 300 baud and higher is 1.

<flow_control> = H*, S, or N – the flow control method: Hardware (CTS/RTS), Software (X–ON, X–OFF) or None.

<physical_interface> = 232*, 422, or 485 – the physical connection protocol for the port: RS–323, RS–422, or RS–485. RS–422 is equivalent to RS–485. All Series 90–30 PCMs support RS–232 only on COM1. IC693PCM300 supports RS–422/485 only on COM2.

With hardware flow control, RTS is turned on when the port is ready to transmit. Then, transmission begins when CTS becomes active. RTS remains on until **<delay value>** expires after the last character is sent.

With software or no flow control, RTS is not turned on, and transmission begins immediately.

<duplex_mode> = 2, 4*, or p – the type of physical connection: 2 = half duplex (2 wire for RS–422/485), 4 = full duplex (4 wire for RS–422/485), p = point–to–point. Available in PCM firmware version 3.00 or later.

In **point–to–point** mode:

- The receiver for the specified port is always enabled.
- When **`<physical_interface>`** = 422 or 485, all RS–485 line drivers for the specified port are enabled when the command is executed and remain on continuously.

In **`full duplex`** mode:

- The receiver for the specified port is always enabled.
- When **`<physical_interface>`** = 422 or 485, the RS–485 line drivers for RTS and transmitted data outputs on the specified port are turned on immediately before transmitting and remain on until **`<delay_value>`** expires after the last character is sent. At all other times, these drivers are in their high–impedance state (tri–stated).

In **half duplex** mode:

- The receiver for the specified port is disabled immediately before transmitting and remains off until **<delay_value>** expires after the last character is sent.
- When **<physical_interface>** = 422 or 485, the RS–485 line drivers for RTS and transmitted data outputs on the specified port are turned on immediately before transmitting and remain on until **<delay_value>** expires after the last character is sent. At all other times, these drivers are in their high–impedance state (tri–stated).

<delay_value> = the time in milliseconds between the end of the last outgoing character and the time RTS is turned off (if applicable), RS–485 line drivers are tri–stated (if applicable), the receiver is enabled in half duplex mode (if applicable), and WAIT mode output statements complete execution. Default = 0.
Available in PCM firmware version 3.00 or later.

<typeahead_size> = the typeahead buffer size in characters for the port. The port can accept up to one less than this number of characters without overflow before an application reads the port. When overflow occurs, any additional characters will be lost. Any size in the range 64 – 32750 bytes may be specified, but the maximum may be limited by available system memory. Default = 320.
Available in PCM firmware version 3.00 or later.

* Default selection.

CPU Setup Strings (special_code = 5)

PLC CPU (*CPU:*) setup strings for *special_code =* 5 have this format:

<PLC_access_password>,<disable_clock_sync>

where:

<PLC_access_password> = the PLC access password for privilege level 2 or higher. If passwords are enabled in the PLC CPU and the PLC has passwords at level 2 and higher, the PCM will be unable to read or write PLC memory until the PCM sends a valid password. Passwords are case sensitive, and valid passwords may have upper case letters, numbers, and underbar ('_') characters only. If an empty string is specified for **<PLC_access_password>**, a password consisting of eight NUL characters will be sent to the PLC CPU. There is no default.

<disable_clock_sync> = N – disables backplane messages the PCM normally sends once per second to synchronize its internal time of day with the PLC CPU. Any character other than 'N' or 'n' enables clock synchronization. Available in PCM firmware version 4.03 or later.

Some applications may be sensitive to the impact that clock synchronization messages have on PLC sweep time or backplane message rates. If these issues are more important than time of day accuracy, use this option. Default = synchronization enabled.

The *notify_code* specifies the method used to notify the calling task that the operation has completed; its value may be WAIT, EVENT_NOTIFY, or AST_NOTIFY. When WAIT is used, there are no *nowait options*, and the return from **Special_dev** is delayed until the operation completes. The other *notify_code*

values cause the function to return immediately, allowing the calling task to continue execution.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Special_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a **far** pointer because its type is not specified by the function prototype in **vtos.h**. If the call succeeds, the **ioreturn** member of the structure at *result_ptr* contains SUCCESS and the **iostatus** member is undefined; when a failure occurs, **ioreturn** contains IO_FAILED, and **iostatus** contains an error status code. For a discussion of asynchronous I/O using event flags, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle* contains a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine,* it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**. If the call succeeds, the **arg2** member of the **ast_blk** contains SUCCESS and the **arg1** member is undefined; when a failure occurs, **arg2** contains IO_FAILED, and **arg1** contains an error status code. For a discussion of asynchronous I/O using AST functions, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

■ Return Value

In WAIT mode, SUCCESS is returned when there are no errors. When an error occurs, IO_FAILED is returned; a status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function call is undefined and may be ignored. Separate return and status values are available in **device_result** and **ast_blk** structures, respectively.

■ See Also

Ioctl_dev

Abort_dev

■ Example

```
#include <vtos.h>
#include <cpu_data.h>
#include <memtypes.h>

msg_addr           other_pcm_addr;
special_dev_8_type new_plc_ref;

word hndl1, hndl2, spec_value;
word task = Get_task_id();

hndl1 = Open_dev( "COM1:", WRITE_MODE, WAIT, task );
hndl2 = Open_dev( "CPU:%R1", READ_MODE | WRITE_MODE, WAIT, task );

spec_value = Special_dev( hndl1, 5, "1200,E,7,1,S,485,2,,512", 0,
                          WAIT, task );
if (spec_value == SUCCESS) {
    /* The serial parameters were changed successfully. */
}

other_pcm_addr.rack = 0;
other_pcm_addr.slot = 5;
other_pcm_addr.svc_point = 8;

spec_value = Special_dev( hndl2, 6, &other_pcm_addr, 0, WAIT, task );

if (spec_value == SUCCESS) {
    /* The channel will send messages to the PCM in rack 0, slot 5. */
    /* The receiving PCM needs to call Open_dev, specifying device  */
    /* "CPU:#8" to receive the messages.  Messages must be 32 bytes */
    /* long and use structure type msg_hdr, defined in CPU_DATA.H,  */
    /* as the low order 16 bytes.                                   */
}

new_plc_ref.type = AI_DATA;
new_plc_ref.offset = 0;

spec_value = Special_dev( hndl2, 8, &new_plc_ref, 0, WAIT, task );
if (spec_value == SUCCESS) {
    /* The channel will now access %AI001 */
}
```

This example opens three I/O channels: one on serial port 1, one to access the PLC register table at %R1, and one to send generic messages to the PLC CPU. **`Special_dev`** is called three times to modify these channels.

The first call resets the serial communication settings for port 1 to 1200 baud, even parity, seven data bits, one stop bit, software flow control, RS-485 interface, 2-wire half duplex operation, and a 512-character type ahead buffer. The turnaround delay is left at the default setting.

The second call changes the rack / slot / service point address for a CPU channel to specify another PCM, rather than the PLC CPU.

The third **`Special_dev`** call sets the channel to access the PLC analog input table at %AI001. Note that the offset member of **`special_dev_8_type`** is zero based.

Start_com_timer

Usage

```
#include <vtos.h>

void Start_com_timer ( timer_handle, count, timeout_routine );
word       timer_handle;
word       count;
void (far* timeout_routine)( void );
```

Description

This function starts a communication timer, specified by *timer_handle*, which was previously allocated by a successful call to **Alloc_com_timer**. The *count* parameter specifies the number of milliseconds which the timer will count before it expires and *timeout_routine* is called. The maximum count value corresponds to 65.535 seconds. A zero (0) *count* causes *timeout_routine* to execute immediately.

The *timeout_routines* may not call most VTOS services, but **Post_ast**, **Iset_ef**, and **Iset_gef** may be called.

Return Value

None.

See Also

Alloc_com_timer, Cancel_com_timer, Dealloc_com_timer

■ Example

```
#include <vtos.h>
#include <stdio.h>

word task;

word far timeout_routine( void )
{
    return( Iset_ef( EF_01, task ) );
}

void main ()
{
    device_result result;
    word com_tmr, local_flags, serial_hndl;
    char buf[265];

    task = Get_task_id();
    Reset_ef( 0xffff );
    com_tmr = Alloc_com_timer();
    serial_hndl = Open_dev( "com1:13", READ_MODE | WRITE_MODE, WAIT, task );

    if (com_tmr != 0) {
        Start_com_timer( com_tmr, 10000, timeout_routine );

        Read_dev( serial_hndl, buf, sizeof( buf ), EVENT_NOTIFY, task, EF_00,
                  (device_result far*)&result );

        Wait_ef( EF_00 | EF_01 );
        local_flags = Test_ef();
        Reset_ef( local_flags );

        if (local_flags & EF_01) {
            /* The timeout occurred. */
            printf( "timeout occurred\n" );
            Abort_dev( serial_hndl, ABORT_ALL, WAIT );
        } else {
            /* Process the received data. */
            Cancel_timer( com_tmr );
            buf[result.ioreturn] = 0;
            printf( "data received: %s\n", (char far*)buf );
        }
        Dealloc_com_timer( com_tmr );
    }
}
```

This example uses a communication timer to limit the wait time for serial input. Serial port 1 is opened with an option that terminates **Read_dev** when a carriage return (ASCII code 13 decimal) character is encountered.

The timer is started before calling **Read_dev** in EVENT_NOTIFY mode. If a carriage return character is read before the timer expires, local event flag **EF_00** is set by the **Read_dev** call. If the timer expires first, event flag **EF_01** is set by **timeout_routine**. In either case, the **Wait_ef** call in **main** returns.

The main program tests its local event flags to determine which event occurred. If a timeout occurred, the **Read_dev** operation is aborted; otherwise the timer is cancelled. Finally, the communication timer is deallocated.

start_plc

Usage

```
#include <cntrl.h>

REQSTAT start_plc ( session_id );
BYTE session_id;
```

Description

This function sets the PLC state to run mode. In Series 90-30 PLCs the output scan is always enabled. For Series 90-70 PLCs, however, the output scan mode is dependent on the CPU hardware switch setting. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

start_plc_noio, start_plc_noio_nowait, start_plc_nowait, stop_plc, stop_plc_nowait

Example

```
#include <cntrl.h>

REQSTAT status;
status = start_plc ( sesn_id );
```

This example uses a WAIT mode request to put the PLC in RUN mode with outputs enabled.

start_plc_noio

Usage

```
#include <cntrl.h>

REQSTAT start_plc_noio ( session_id );
BYTE session_id;
```

Description

This function sets the state of a Series 90-70 PLC to run mode with I/O disabled, even if the hardware switch on the CPU module is in the I/O ENABLE position. This request is valid only for Series 90-70 PLCs. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

start_plc, start_plc_noio_nowait, start_plc_nowait, stop_plc, stop_plc_nowait

Example

```
#include <cntrl.h>

REQSTAT status;
status = start_plc_noio ( sesn_id );
```

This example uses a WAIT mode request to put the PLC in RUN mode with outputs disabled.

start_plc_noio_nowait

Usage

```
#include <cntrlnw.h>

REQID start_plc_noio_nowait ( session_id );
BYTE session_id;
```

Description

See **start_plc_noio**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

reqstatus, start_plc, start_plc_noio, start_plc_nowait, stop_plc, stop_plc_nowait

Example

```
#include <cntrlnw.h>

REQID request_id;
REQSTAT status;

request_id = start_plc_noio_nowait ( sesn_id );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC is running with outputs disabled */
}
```

This example uses a NOWAIT mode request to put the PLC in RUN mode with outputs disabled.

start_plc_nowait

Usage

```
#include <cntrlnw.h>

REQID start_plc_nowait ( session_id );
BYTE session_id;
```

Description

This function sets the PLC state to run mode. For Series 90-30 PLCs, the output scan is always enabled. For Series 90-70 PLCs, however, the output scan mode is dependent on the CPU hardware switch setting. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

reqstatus, start_plc, start_plc_noio, start_plc, start_plc_noio_nowait, stop_plc, stop_plc_nowait

Example

```
#include <cntrlnw.h>

REQID request_id;
REQSTAT status;

request_id = start_plc_nowait ( sesn_id );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC is running with outputs enabled */
}
```

This example uses a NOWAIT mode request to put the PLC in RUN mode with outputs enabled.

Start_timer

Usage

```
#include <vtos.h>

word Start_timer ( timer_code, hi_cnt, mid_cnt,
                   low_cnt, <notify option> );
word timer_code;
word hi_cnt;
word mid_cnt;
word low_cnt;
```

where **<***notify option***>** depends on the value of *timer_code*:

```
word Start_timer ( AST_NOTIFY_MODE | <other codes>, hi_cnt,
                   mid_cnt, low_cnt, ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;

word Start_timer ( <other codes>, hi_cnt, mid_cnt,
                   low_cnt, event_flag_mask );
word               event_flag_mask;
```

Description

This function starts a general purpose timer. The *timer_code* consists of one or more timer control codes OR-ed together, as shown in this table.

Control Code	Description
RELATIVE_TIMEOUT	The timer will expire at a future time calculated by adding the current time of day to the time interval specified by *hi_cnt, mid_cnt*, and *low_cnt*. This is the only timeout mode supported by the current implementation; it **must** be specified.
REPEAT_MODE	The timer will restart when it expires. If REPEAT_MODE is not specified, one-shot operation will occur.
TASK_SPECIFIED	This code specifies that VTOS should notify a task specified by the task value bits. If this bit is not set, the calling task will be notified.
AST_NOTIFY_MODE	This code specifies that VTOS should notify the calling task when the timer expires by posting an asynchronous trap (AST).
Task value	When TASK_SPECIFIED is used in timer_code, the task number value for the target task must be OR-ed into *timer_code*. The specified task will be notified using one or more of its local event flags, as specified in *event_flag_mask*, or by the specified *ast_routine*. When the special task value **0xFF** (-1 expressed as a **byte**) is specified, the global event flags specified in *event_flag_mask* will be set when the timer expires. Any tasks waiting for these flags will be notified.

The timeout value for **Start_timer** may contain any time duration in the range from zero time to the largest number of milliseconds which can be expressed as a long unsigned integer: 49 days, 17 hours, 2 minutes, 47 seconds, 295 milliseconds. It may be specified either as milliseconds or clock time (days, hours, minutes, seconds, milliseconds) format. This table shows the content of the three count parameters for both formats.

Count Parameter	Format	Content
hi_cnt	Milliseconds.	MS_COUNT_MODE.
	Clock time.	A **word** value which contains the number of days in the most significant byte and hours in the least significant byte.
mid_cnt	Milliseconds.	A **word** value which contains the *most* significant word of the long unsigned integer millisecond count.
	Clock time.	A **word** value which contains the number of minutes in the most significant byte and seconds iu the least significant byte.
low_cnt	Milliseconds.	A **word** value which contains the *leas*t significant word of the long unsigned integer millisecond count.
	Clock time.	A **word** value which contains the number of milliseconds.

When *timer_code* includes AST_NOTIFY_MODE, the *ast_routine* parameter must contain the address of an asynchronous trap (AST) handler function defined in the code for the task to be notified. The optional *ast_handle* contains a user-selected tag value to identify the timeout event, if necessary, for the AST function.

When *timer_code* does not include AST_NOTIFY_MODE, the *event_flag_mask* parameter contains bits which specify one or more event flags. The least significant byte of *timer_code* must specify a task to be notified. If the task value corresponds to an actual application task, one or more of its local event flags will be set when the timer expires, as specified in *event_flag_mask*. If the task value contains **0xFF**, the global event flags specified by *event_flag_mask* are set when the timer expires.

■ Return Value

A timer handle value or zero (0) is returned. Valid timer handles are non-zero unsigned integers. The handle may be used to identify the timer in calls to **Cancel_timer**. When an error occurs, **Start_timer** returns zero (0), and an error code from this table is in **_VTOS_error**.

Error Code	Description
BAD_OPCODE	The *timer_code* is invalid.
NO_TIMERS	All the general purpose timers are being used.

See Also

Cancel_timer, Wait_time

Example

```
#include <vtos.h>

void far timer_ast_func ( ast_blk far* p )
{
/*
 * Process the timeout event identified by p->handle.  The arg1 .. arg4
 * members are undefined.
 */
}

void main ()
{
    word h1, h2, h3, h4, h5;

    h1 = Start_timer ( RELATIVE_TIMEOUT | TASK_SPECIFIED | 7,
                MS_COUNT_MODE, 0, 500, EF_02 );

    if (!h1) {
       /* _VTOS_error contains the error code */
    }
    h2 = Start_timer ( RELATIVE_TIMEOUT | TASK_SPECIFIED | 0xFF,
                MS_COUNT_MODE, 0, 500, EF_01 | EF_15 );

    h3 = Start_timer ( RELATIVE_TIMEOUT | TASK_SPECIFIED | 14,
                (6 << 8) | 23, (59 << 8) | 59, 999, EF_00 );

    h4 = Start_timer ( RELATIVE_TIMEOUT | AST_NOTIFY_MODE,
                MS_COUNT_MODE, 0, 500, timer_ast_func, 1 );

    h5 = Start_timer ( RELATIVE_TIMEOUT | AST_NOTIFY_MODE,
                 0, (2 << 8), 0, 1, timer_ast_func, 2 );

    Cancel_timer (h3);
}
```

This example starts five timers, identified by the handles **h1**, **h2**, **h3, h4**, and **h5,** respectively. The first timer runs for 500 milliseconds. When it expires, local event flag **EF_02** is set for task 7, which then becomes ready to run. Handle h1 is checked to ensure that the timer was actually started. The second call also starts a 500 millisecond timer, and global event flags **EF_01** and **EF_15** are set when it expires. The third call notifies task 14 via local event flag **EF_00** when the timer expires. The timer is set for six days, 23 hours, 59 minutes, 59 seconds and 999 milliseconds. The fourth call notifies the calling task with an AST when the 500 millisecond timer expires. The AST function, **timer_ast_func**, is also defined in the example. An AST handle value of one (1) identifies the timeout event to **timer_ast_func.** The last call also notifies the calling task through **timer_ast_func** when the two minute timeout expires. A different AST handle value is used so that **timer_ast_func** can distinguish the two events. Finally, timer handle **h3** is cancelled.

stop_plc

Usage

```
#include <cntrl.h>

REQSTAT stop_plc ( session_id );
BYTE session_id;
```

Description

This function sets the PLC state to stop mode. When the execution sweep stops, the I/O scan may or may not continue, depending on the IOScan-Stop software configuration item for the PLC CPU in the Logicmaster 90 I/O configuration. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

start_plc, start_plc_noio, start_plc_noio_nowait, start_plc_nowait, stop_plc_nowait

Example

```
#include <cntrl.h>

REQSTAT status;
status = stop_plc ( sesn_id );
```

This example uses a WAIT mode request to stop the PLC.

stop_plc_nowait

Usage

```
#include <cntrlnw.h>

REQID stop_plc_nowait ( session_id );
BYTE session_id;
```

Description

See **stop_plc**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

reqstatus, start_plc, start_plc_noio, start_plc_noio_nowait, start_plc_nowait, stop_plc

Example

```
#include <cntrlnw.h>

REQID request_id;
REQSTAT status;request_id = stop_plc_nowait ( sesn_id );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the PLC was stopped */
}
```

This example uses a NOWAIT mode request to stop the PLC.

Suspend_task

Usage

```
#include <vtos.h>

void Suspend_task ( task_id );
word task_id;
```

Description

Suspend_task prevents a task from executing until it is resumed by calling **Resume_task**. When a task is suspended, all VTOS activity which it previously initiated, such as serial I/O, will continue, but the task itself will no longer run. A task may suspend itself; if so, it must be resumed by another task.

A task may be suspended more than once: for example, by several different tasks. VTOS maintains a count of the number of times each task has been suspended. The count is incremented on each **Suspend_task** call for a given *task_id* and decremented on each **Resume_task** call for the same *task_id*. The suspended task does not become active until the count reaches zero.

If a task is suspended while it is waiting for I/O, asynchronous traps (ASTs), event flags, or a timer, it will execute only after the event it is waiting for has occurred **and** it has been resumed.

Caution

When a task which receives asynchronous traps (ASTs) regularly is suspended, ast_blk structures for the task will accumulate until the task is resumed. These structures can exhaust free memory if the task remains suspended indefinitely, causing the PCM to lock up.

Return Value

None.

See Also

Resume_task

Example

```
#include <vtos.h>

Suspend_task ( 15 );
```

This example suspends task 15, which will not execute until it is resumed by calling **Resume_task**.

terminate_comm_session

Usage

```
#include <session.h>

REQSTAT terminate_comm_session ( session_id );
BYTE session_id;
```

Description

This function terminates a communication session which was previously opened by calling **api_initialize** and **establish_comm_session**. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize, configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

api_initialize, establish_comm_session

Example

See **api_initialize**.

Terminate_task

Usage

```
#include <vtos.h>

void Terminate_task ( task_id );
word task_id;
```

Description

Terminate_task causes the task specified by *task_id* to be permanently de-activated. The effect of **Terminate_task** is irreversible. The specified task's resources (memory, open communication channels, timers and pending ASTs) are returned to VTOS. However, code space in RAM for the task is not de-allocated. A task can terminate itself or any other task. When a task terminates itself, there is no return from the call. VTOS itself calls **Terminate_task** when a task exits its **main** function.

Caution

DO NOT terminate any of the PCM system tasks, with Task ID values zero through three. PCM lockup or unexpected operation will result.

Return Value

None.

See Also

Init_task, Process_env

Example

```
#include <vtos.h>

Terminate_task ( Get_task_id () );
```

In this example, the calling task terminates itself.

Test_ef

Usage

```
#include <vtos.h>

word Test_ef ( void );
```

Description

This function is used by the calling task to determine which, if any, of its local event flags are set. There are no parameters. There is no mechanism for a task to test the local event flags of a different task.

Note

The result returned by **Test_ef** may not be valid by the time the calling task decides what action to take following the call. Higher priority tasks or interrupts can set additional event flags.

Return Value

A word is returned which contains the status of all the calling task's local event flags.

See Also

Iset_ef, Reset_ef, Set_ef, Wait_ef

Example

```
#include <vtos.h>

word ef;

ef = Test_ef ();
Reset_ef ( ef );
/* Check the individual flags in ef. */

}
```

In this example, the calling task calls **Test_ef** to look at its local event flags. The flags which were set are reset. The task can now check the flags which were set.

Test_gef

Usage

```
#include <vtos.h>

word Test_gef ( void );
```

Description

This function is used to determine which, if any, global event flags are set. There are no parameters. Any task may reset, set, and test global event flags.

Note

The result returned by **Test_gef** may not be valid by the time the calling task decides what action to take following the call. Higher priority tasks or interrupts can set additional event flags or reset the event flag of interest.

Return Value

A word is returned which contains the status of all global event flags.

See Also

Iset_gef, Reset_gef, Set_gef, Wait_gef

Example

```
#include <vtos.h>

Reset_gef ( EF_12 );

while (!(Test_gef () & EF_12)) {
    /* repeat some action until stopped by another task */
}
```

The calling task of this example resets a global event flag and then performs some repetitive processing until a different task stops it by setting the event flag. Note that the operation might never occur; a higher priority task could pre-empt this one and set the flag between the **Reset_gef** and **Test_gef** calls.

Test_task

Usage

```
#include <vtos.h>

word Test_task ( void );
```

Description

This function is used to determine which VTOS tasks have been started but not terminated. It can be used to determine available task numbers, for the purposes of installing new tasks.

Return Value

A word value is returned in which each bit which is set indicates a task number in use.

See Also

Example

```
#include <vtos.h>
#include <stdio.h>

word task_masks [] = {
    TASK_00_MASK, TASK_01_MASK, TASK_02_MASK, TASK_03_MASK,
    TASK_04_MASK, TASK_05_MASK, TASK_06_MASK, TASK_07_MASK,
    TASK_08_MASK, TASK_09_MASK, TASK_10_MASK, TASK_11_MASK,
    TASK_12_MASK, TASK_13_MASK, TASK_14_MASK, TASK_15_MASK
};

int i;

word active_tasks;
active_tasks = Test_task ();
for ( i = 0; I < NUM_PCM_TASKS; ++I ) {
    if ( active_tasks & task_masks [i] )
       printf ( "task %d is active\n", i );
}
```

This example prints the task numbers of all the currently active PCM tasks.

Unblock_sem

Usage

```
#include <vtos.h>

void Unblock_sem ( semaphore_handle );
word semaphore_handle;
```

Description

This function is called to release a semaphore, specified by *semaphore_handle*, which the calling task acquired by calling either **Link_sem** or **Block_sem**. Semaphores **must** be unblocked before any other VTOS function call is made.

Return Value

None.

See Also

Block_sem, Link_sem, Unlink_sem

Example

See **Block_sem**.

Unlink_sem

Usage

```
#include <vtos.h>

void Unlink_sem ( semaphore_handle );
word semaphore_handle;
```

Description

This function is called when a task no longer requires the use of a particular semaphore, specified by *semaphore_handle*. The handle must be a value returned by a previous, successful call to **Link_sem**. If no other tasks are linked to the semaphore, its memory block is released to free memory. This function is rarely used.

Return Value

None.

See Also

Block_sem, Link_sem, Unblock_sem

Example

See **Block_sem**.

update_plc_status

Usage

```
#include <utils.h>

REQSTAT update_plc_status ( session_id );
BYTE session_id;
```

Description

This function requests an update of the PLC status information record in the **plc_status_info** array corresponding to the specified *session_id*. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. Note that all other PLC requests also update this information. This function has a minimal effect on PLC sweep time, and is most often used when no other PLC data is required.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

Example

```
#include <utils.h>

REQSTAT status;
status = update_plc_status ( sesn_id );
```

This example uses a WAIT mode request to update the PLC status data in the PCM.

update_plc_status_nowait

Usage

```
#include <utilsnw.h>

REQID update_plc_status_nowait ( session_id );
BYTE session_id;
```

Description

See **update_plc_status**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

reqstatus

Example

```
#include <utilsnw.h>

REQID request_id;
REQSTAT status;request_id = update_plc_status_nowait ( sesn_id );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the new PLC status data is available */
}
```

This example uses a NOWAIT mode request to update the PLC status data in the PCM.

Vme_clear_lcl_sem

Usage

```
#include <vme.h>

int _cdecl Vme_clear_lcl_sem( void );
```

Description

`Vme_clear_lcl_sem` is used by Series 90–70 PCM applications to clear a semaphore in the VME memory of the host PCM after the application has acquired the semaphore with `Vme_test_lcl_sem` and accessed the memory it controls.

`Vme_clear_lcl_sem` has no parameters.

`Vme_clear_lcl_sem` is provided in PCM C toolkit versions 1.05 and later. Attempting to use it with earlier versions of the toolkit will cause a linkage error (unresolved external reference).

Return Value

`Vme_clear_lcl_sem` returns a value from this table.

Return Value	Completion Status
SUCCESS	A previously acquired semaphore was cleared successfully.
NO_SEM_PREV_ACQUIRED	No semaphore has been acquired in VME memory.

See Also

`Vme_read, Vme_test_and_set, Vme_test_lcl_sem, Vme_write`

Example

See `Vme_test_lcl_sem`.

Vme_read

Usage

```
#include <vme.h>

int _cdecl Vme_read( void far *dest,
                     byte      vme_hi_adr,
                     word      vme_lo_adr,
                     byte      vme_am_code,
                     long      len,
                     byte      units );
```

Description

Vme_read is used by Series 90–70 PCM applications to read VMEbus memory in other modules in the Series 90–70 PLC. The PCM performs the memory read as a VMEbus master.

The vme_am_code parameter contains either STD_NON_PRIV or one of the values SHORT_NP_RACK0 through SHORT_NP_RACK7, depending on the rack location of the target module. When vme_am_code contains STD_NON_PRIV , vme_hi_addr must contain the most significant 8 bits of the standard non–privileged (A24) VMEbus address; otherwise it is ignored. The vme_lo_addr parameter contains either the least significant 16 bits of a standard non–privileged address or the entire short non–privileged (A16) address when vme_am_code contains one of the values SHORT_NP_RACK0 through SHORT_NP_RACK7.

The dest parameter specifies the **far** address of a memory buffer where data will be copied to, units contains either BYTE_UNITS or WORD_UNITS, and len specifies the number of bytes or words to copy. However, if vme_lo_addr specifies an odd start address, the units parameter will be silently ignored, and BYTE_UNITS will be used for the data transfer.

Vme_read is provided in PCM C toolkit versions 1.05 and later. Attempting to use it in earlier versions of the toolkit will cause a linkage error (unresolved external reference).

Return Value

Vme_read returns a value from this table.

Return Value	_VTOS_error	Completion Status
SUCCESS	SUCCESS	The specified VME read completed successfully.
IO_FAILED	INVALID_PCM	The PCM does not support VME transfer functions – it is either a Series 90–30 PCM or a Series 90–70 PCM with PCMA1 hardware or firmware earlier than version 4.03
	VMEbus error bits	. BUSGRT_MASK is set in _VTOS_error when the PLC CPU refuses to grant the PCM access to the VMEbus. BSERR_MASK or–ed with BUSHOG_MASK is returned when there is no physical memory at the specified VMEbus address, and when a PCM attempts to read a VMEbus address located in its own VME memory.

See Also

Vme_clear_lcl_sem, Vme_test_and_set, Vme_test_lcl_sem, Vme_write

Example

```
#include <vme.h>
#include <stdio.h>

byte read_data[32];
int status;
byte hi_adr;
byte rack = 0;
byte slot = 3;

if (!S9070_RACKSLOT_VALID(rack, slot)) {
    /* handle the error */
}

hi_adr = S9070_VME_HI_ADDR(rack, slot);
status = Vme_read( read_data, hi_adr, 0, STD_NON_PRIV, 32, BYTE_UNITS );

if (!status) {
    int c, i, j;
    char buf[20];

    for (i = VME_DATA_LEN; i; ) {
        printf( "%04x:  ", (VME_DATA_LEN - i) );

        for (j=16; i&&j; --j, --i) {
            c = read_data[VME_DATA_LEN - i];
            buf[16-j] = ((c >= ' ' && c <= '~') ? c : '.');
            printf( "%02x ", c );
        }

        buf[16-j] = 0;

        for ( ; j; --j) {
            printf( "   " );
        }

        printf( " \"%s\"\n", (char far* )buf );
    }
)
```

This example reads 32 bytes from the start of VME memory in the module in rack zero, slot 3. The most significant byte of the VME address is calculated by the S9070_VME_HI_ADDR macro in VME.H.

When the target module is a Series 90–70 PCM, the program should print this output:

```
0000:  20 56 20 4d 20 45 20 49 20 44 20 47 20 45 20 46  " V M E I D G E F"
0010:  20 37 20 50 20 43 20 4d 20 37 20 31 20 31 20 41  " 7 P C M 7 1 1 A"
```

Vme_test_and_set

Usage

```
#include <vme.h>

int _cdecl Vme_test_and_set( byte vme_hi_adr,
                             word vme_lo_adr,
                             byte vme_am_code,
                             byte units );
```

Description

`Vme_test_and_set` is used by Series 90–70 PCM applications to control access to VMEbus memory of other modules in its Series 90–70 PLC using a semaphore. It works like the VMETS function block for Series 90–70 PLC CPUs. The PCM, operating as a VMEbus master, uses a locked memory exchange to test the byte or word semaphore at a specified VMEbus address.

Typically, PCM applications that use shared VMEbus memory provide a semaphore to control access to that memory. The PCM whose memory is shared uses `Vme_test_lcl_sem` and `Vme_clear_lcl_sem` to access the semaphore locally, while other PCMs use `Vme_test_and_set`.

The `vme_am_code` parameter contains either STD_NON_PRIV or one of the values SHORT_NP_RACK0 through SHORT_NP_RACK7, depending on the rack location of the target module. When `vme_am_code` contains STD_NON_PRIV , `vme_hi_addr` must contain the most significant 8 bits of the standard non–privileged (A24) VMEbus address; otherwise it is ignored. The `vme_lo_addr` parameter contains either the least significant 16 bits of a standard non–privileged address or the entire short non-privileged (A16) address when `vme_am_code` contains one of the values SHORT_NP_RACK0 through SHORT_NP_RACK7.

The `units` parameter contains either BYTE_UNITS or WORD_UNITS and specifies whether the semaphore is a byte or word variable. However, if `vme_lo_addr` specifies an odd start address, the units parameter will be silently ignored, and BYTE_UNITS will be used for the data transfer.

`Vme_test_and_set` is provided in PCM C toolkit versions 1.05 and later. Attempting to use it in earlier versions of the toolkit will cause a linkage error (unresolved external reference).

Return Value

`Vme_test_and_set` returns a value from this table.

Return Value	_VTOS_error	Completion Status
SEM_ACQUIRED	SUCCESS	The PCM acquired the specified semaphore and the memory it controls.
SEM_NOT_ACQUIRED	SUCCESS	The VME operation completed successfully, but the semaphore was not acquired.
IO_FAILED	INVALID_PCM	The PCM does not support VME transfer functions – it is either a Series 90-30 PCM or a Series 90-70 PCM with PCMA1 hardware or firmware earlier than version 4.03.
	VMEbus error bits	BUSGRT_MASK is set in _VTOS_error when the PLC CPU refuses to grant the PCM access to the VMEbus. BSERR_MASK or–ed with BUSHOG_MASK is returned when there is no physical memory at the specified VMEbus address, and when a PCM attempts to read a VMEbus address located in its own VME memory.

See Also

Vme_clear_lcl_sem, Vme_read, Vme_test_lcl_sem, Vme_write

Example

```
#include <vme.h>
#include <stdio.h>

word sem_copy;
int status;
byte hi_adr;
byte rack = 0;
byte slot = 3;

if (!S9070_RACKSLOT_VALID(rack, slot)) {
    /* handle the error */
}

hi_adr = S9070_VME_HI_ADDR(rack, slot);
status = Vme_test_and_set( hi_adr, 0x4000, STD_NON_PRIV, WORD_UNITS );

if (!_VTOS_error)
    if (status)  {
        /* the semaphore was acquired – access the data it controls */
        /* then use VME_write() to release the semaphore */
        sem_copy = 0;
        VME_write(hi_adr, 0x4000, STD_NON_PRIV, &sem_copy, 1, WORD_UNITS );
    } else {
        /* the semaphore was not acquired – try again later */
    }
} else {
    /* _VTOS_error indicates an error occurred – process the error */
}
```

This example attempts to acquire a semaphore in VME memory. The most significant byte of the VME address is calculated by the S9070_VME_HI_ADDR macro in VME.H.

Vme_test_lcl_sem

Usage

```
#include <vme.h>

int _cdecl Vme_test_lcl_sem( word local_vme_offset, byte units );
```

Description

Vme_test_lcl_sem is used by Series 90–70 PCM applications to acquire a semaphore in the VME memory of the host PCM in order to access the memory it controls. This function, along with Vme_clear_lcl_sem, enables sharing VME memory in the local PCM with applications in the PLC CPU or in other PCMs. CPU applications would use VMETS and VMEWRT function blocks to test and clear the this semaphore, while applications in other PCMs would use Vme_test_and_set and Vme_write.

The local_vme_offset parameter is the VME memory offset of a semaphore in the host PCM, and units contains either BYTE_UNITS or WORD_UNITS. However, if local_vme_offset specifies an odd address, the units parameter will be silently ignored, and BYTE_UNITS will be used for the semaphore operation.

Vme_test_lcl_sem is provided in PCM C toolkit versions 1.05 and later. Attempting to use it with earlier versions of the toolkit will cause a linkage error (unresolved external reference).

Return Value

`Vme_test_lcl_sem` returns a value from this table.

Return Value	Completion Status
SEM_NOT_ACQUIRED	The specified semaphore was not acquired.
SEM_ACQUIRED	The specified semaphore was acquired successfully.
INVALID_PCM	The PCM does not support VME transfer functions – it is either a Series 90–30 PCM or a Series 90–70 PCM with PCMA1 hardware or firmware earlier than version 4.03.
INVALID_ADDR	The `local_vme_offset` parameter is invalid. It is either in memory used by the PLC CPU, or the PCM uses PCMA1 hardware and the specified offset is inaccessible.
TOO_MANY_SEMS	Only one semaphore can be acquired at one time, and one is already in use.

See Also

`Vme_clear_lcl_sem, Vme_read, Vme_test_and_set, Vme_write`

■ Example

```
#include <vme.h>
#include <stdio.h>

typedef struct {
    word sem;
    word data[31];
} vme_data_t;

vme_data_t far* vme_data_ptr;
int status;

FP_SEG( vme_data_ptr ) = VME_DP_SEG;
FP_OFF( vme_data_ptr ) = 0x4010;

status = Reserve_dp_buff( vme_data_ptr, sizeof( vme_data_t ) );

if (status == SUCCESS) {
    status = Vme_test_lcl_sem( FP_OFF( vme_data_ptr ), WORD_UNITS );
}

if (status == SEM_ACQUIRED) {
    /* access the data */
    Vme_clear_lcl_sem();
}
```

This example reserves a block of VME memory. Then it acquires control of the memory block using `Vme_test_lcl_sem` and accesses the data in the block. Finally, control of the block is released by calling `Vme_clear_lcl_sem`.

Vme_write

Usage

```
#include <vme.h>

int _cdecl Vme_write( byte      vme_hi_adr,
                      word      vme_lo_adr,
                      byte      vme_am_code,
                      void far *src,
                      long      len,
                      byte      units );
```

Description

`Vme_write` is used by Series 90–70 PCM applications to write VMEbus memory in other modules in the Series 90–70 PLC. The PCM performs the memory write as a VMEbus master.

The `vme_am_code` parameter contains either STD_NON_PRIV or one of the values SHORT_NP_RACK0 through SHORT_NP_RACK7, depending on the rack location of the target module. When `vme_am_code` contains STD_NON_PRIV , `vme_hi_addr` must contain the most significant 8 bits of the standard non–privileged (A24) VMEbus address; otherwise it is ignored. The `vme_lo_addr` parameter contains either the least significant 16 bits of a standard non–privileged address or the entire short non–privileged (A16) address when `vme_am_code` contains one of the values SHORT_NP_RACK0 through SHORT_NP_RACK7.

The `src` parameter specifies the **far** address of a memory buffer where data will be copied from, `units` contains either BYTE_UNITS or WORD_UNITS, and len specifies the number of bytes or words to copy. However, if `vme_lo_addr` specifies an odd start address, the `units` parameter will be silently ignored, and BYTE_UNITS will be used for the data transfer.

`Vme_write` is provided in PCM C toolkit versions 1.05 and later. Attempting to use it in earlier versions of the toolkit will cause a linkage error (unresolved external reference).

Return Value

`Vme_write` returns a value from this table.

Return Value	_VTOS_error	Completion Status
SUCCESS	SUCCESS	The specified VME write completed successfully.
IO_FAILED	INVALID_PCM	The PCM does not support VME transfer functions – it is either a Series 90-30 PCM or a Series 90-70 PCM with PCMA1 hardware or firmware earlier than version 4.03.
	VMEbus error bits	BUSGRT_MASK is set in _VTOS_error when the PLC CPU refuses to grant the PCM access to the VMEbus. BSERR_MASK or–ed with BUSHOG_MASK is returned when there is no physical memory at the specified VMEbus address, and when a PCM attempts to read a VMEbus address located in its own VME memory.

See Also

Vme_clear_lcl_sem, Vme_test_and_set, Vme_test_lcl_sem, Vme_read

Example

```
#include <vme.h>
#include <stdio.h>

byte write_data[32] = {
    48, 49, 50, 51, 51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63,
    64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
};
byte read_data[32] = { 0 };
int status;
byte hi_adr;
byte rack = 0;
byte slot = 3;

if (!S9070_RACKSLOT_VALID(rack, slot)) {
    /* handle the error */
}

hi_adr = S9070_VME_HI_ADDR(rack, slot);
status = Vme_write( hi_adr, 0x4000, STD_NON_PRIV, write_data, 16, WORD_UNITS );

if (!status) {
    status = Vme_read( read_data, hi_adr, 0x4000, STD_NON_PRIV, 32, BYTE_UNITS );
}

if (!status) {
    int c, i, j;
    char buf[20];

    for (i = VME_DATA_LEN; i; ) {
        printf( "%04x:  ", (VME_DATA_LEN - i) );

        for (j=16; i&&j; --j, --i) {
            c = read_data[VME_DATA_LEN - i];
            buf[16-j] = ((c >= ' ' && c <= '~') ? c : '.');
            printf( "%02x ", c );
        }

        buf[16-j] = 0;

        for ( ; j; --j) {
            printf( "   " );
        }

        printf( " \"%s\"\n", (char far* )buf );
    }
}
```

This example writes 16 words of data to offset 4000 hexadecimal of VME memory in the module in rack zero, slot 3. The most significant byte of the VME address is calculated by the S9070_VME_HI_ADDR macro in VME.H. The same data is read back to a different buffer as bytes. When the target module has VME memory at the specified address, the program should print this output

```
0000:  30 31 32 33 33 35 36 37 38 39 3a 3b 3c 3d 3e 3f  "0123356789:;<=>?"
0010:  40 41 42 43 44 45 46 47 48 49 4a 4b 4c 4d 4e 4f  "@ABCDEFGHIJKLMNO"
```

Wait_ast

Usage

```
#include <vtos.h>

void Wait_ast ( void );
```

Description

`Wait_ast` suspends execution of the calling task until an asynchronous trap (AST) is received.

Caution

If the event which posts the expected AST has already occurred, and no other ASTs are posted, the calling task will never resume execution.

The example below shows how to avoid this problem.

Return Value

None.

See Also

`Post_ast`

■ Example

```
#include <vtos.h>
#include <dos.h>

word event_occurred;

void far ast_func ( ast_blk far* p )
{
    event_occurred = TRUE;
/* process the event */
}

void main ()
{
    event_occurred = FALSE;
/*
 * Perform some operation with AST notification;
 * specify ast_func as the AST handler function.
 */
    _disable ();
    if (!event_occurred) {
       Wait_ast ();
    }
    _enable ();
}
```

This example waits for an AST only when the expected event has not already occurred. The **_disable** and **_enable** functions are from the Microsoft C runtime library. They disable and enable maskable interrupts, respectively, but only within the task's main execution thread. If the call to **_enable** is omitted, erratic operation will result.

Wait_ef

Usage

```
#include <vtos.h>

void Wait_ef ( local_ef_mask );
word local_ef_mask;
```

Description

This function is called to wait for one or more of the calling task's sixteen local event flags; *local_ef_mask* specifies the flag or flags to wait for. If one of the specified event flags is already set when **Wait_ef** is called, control returns immediately to the calling task. Otherwise, the task remains suspended until at one of the specified event flag is set. During this time, the task will be able to receive ASTs, but it will not be able to process them. All the task's event flags remain unchanged.

A task's local event flags may be set by any task, but only the task in which they are local may test or reset them.

Return Value

None.

See Also

Iset_ef, Iset_gef, Reset_ef, Reset_gef, Set_ef, Set_gef, Test_ef, Test_gef, Wait_gef

■ Example

```
#include <vtos.h>

Reset_ef ( EF_00 | EF_12 );
/*
 * Perform an EVENT_NOTIFY I/O operations using EF_00 for notification.
 * Start a timer using EF_12 for notification.
 */
Wait_ef ( EF_00 | EF_12 );
if ( Test_ef () & EF_00 ) {
/*
 * the I/O operation completed
 */
} else {
/*
 * the I/O operation timed out before completion
 */
}
```

In this example, the calling task waits for either **EF_00** or **EF_12** to be set. Then it tests it local event flags to determine whether **EF_00** was the one which resumed its execution.

Wait_gef

Usagae

```
#include <vtos.h>

void Wait_gef ( global_ef_mask );
word global_ef_mask;
```

Description

This function is called to wait for one or more of the sixteen global event flags; *global_ef_mask* specifies the flag or flags to wait for. If one of the specified event flags is already set when **Wait_gef** is called, control returns immediately to the calling task. Otherwise, the task remains suspended until at one of the specified event flag is set. During this time, the task will be able to receive ASTs, but it will not be able to process them. All the task's event flags remain unchanged.

Global event flags may be set, reset, or tested by any task.

Return Value

None.

See Also

Iset_ef, Iset_gef, Reset_ef, Reset_gef, Set_ef, Set_gef, Test_ef, Test_gef, Wait_ef

Example

```
#include <vtos.h>

Reset_gef ( EF_02 );
Wait_gef ( EF_02 );
```

The task in this example resets global event flag **EF_02** and then waits for another task to set it.

Wait_task

Usage

```
#include <vtos.h>

void Wait_task ( task_mask );
word task_mask;
```

Description

Wait_task suspends the calling task until one or more lower priority tasks, specified by *task_mask*, have terminated. It may be used by a main task which starts one or more temporary tasks and then waits for them to complete their operation before it continues.

Return Value

None.

See Also

Init_task, Process_env

Example

```
#include <vtos.h>

Wait_task ( TASK_15_MASK );
```

The task in this example suspends itself until task number 15 terminates.

Wait_time

Usage

```
#include <vtos.h>

void Wait_time ( hi_cnt, mid_cnt, low_cnt );
word hi_cnt;
word mid_cnt;
word low_cnt;
```

Description

When this function is called, the calling task is suspended until the time specified by *hi_cnt*, *mid_cnt*, and *low_cnt* has expired. Any time duration which can be expressed as a long unsigned integer number of milliseconds may be specified: zero to 49 days, 17 hours, 2 minutes, 47 seconds, 295 milliseconds. The time value may be specified either as milliseconds or clock time (days / hours / minutes / seconds / milliseconds) format. This table shows the content of the three count parameters for both formats.

Count Parameter	Format	Content
hi_cnt	Milliseconds.	MS_COUNT_MODE.
	Clock time.	A **word** value which contains the number of days in the most significant byte and hours in the least significant byte.
mid_cnt	Milliseconds.	A **word** value which contains the *most* significant word of the long unsigned integer millisecond count.
	Clock time.	A **word** value which contains the number of minutes in the most significant byte and seconds iu the least significant byte.
low_cnt	Milliseconds.	A **word** value which contains the *least* significant word of the long unsigned integer millisecond count.
	Clock time.	A **word** value which contains the number of milliseconds.

Return Value

`Wait_time` has no return value. If an error occurs, an error code from this table will be in **`_VTOS_error`**.

Error Code	Description
NO_TIMERS	Allthe general purpose timers are being used.

See Also

`Start_timer`

Example

```
#include <vtos.h>

Wait_time ( MS_COUNT_MODE, 0, 500 );
Wait_time ( (6 << 8) | 23, (59 << 8) | 59, 999 );
```

In this example, the calling task waits for 500 milliseconds. Then it waits for six days, 23 hours, 59 minutes, 59 seconds and 999 milliseconds.

Where_am_i

Usage

```
#include <vtos.h>

long Where_am_i ( void );
```

Description

This function returns the PLC rack / slot location in which the PCM where it executes is installed.

This function is available in PCM version 3.00 and later revisions.

Return Value

When the call succeeds, **`Where_am_i`** returns a long integer which contains a Series 90 PLC rack / slot address in the form used as the source and destination addresses of generic backplane messages. The example below shows how to extract the rack, slot and service point values.

`Where_am_i` does not know the PCM location for a short time after VTOS is initialized. During this time, LONG_FAILURE is returned.

See Also

Example

```
#include <vtos.h>
#include <stdio.h>

long location;
int rack, slot, svc_pt;

location = Where_am_i ();
rack = location & 0x000F;
slot = (location >> 4) & 0x001F;
svc_pt = (location >> 9) & 0x003F;

printf ( "PCM location is:  rack %d, slot %d, service point %d\n",
       rack, slot, svc_pt );
```

This example prints the PCM location to the STDOUT device.

Write_dev

Usage

```
#include <vtos.h>

word Write_dev ( device_handle, buffer, size, notify_code,
                 task_id [, <nowait options>] );
word      device_handle;
void far* buffer;
word      size;
word      notify_code;
word      task_id;
```

where **<***nowait options***>** depend on the value of *notify_code*:

```
word Write_dev ( device_handle, buffer, size, WAIT, task_id );

word Write_dev ( device_handle, buffer, size, EVENT_NOTIFY,
                 task_id, local_ef_mask,
                (device_result far*)result_ptr );
word              local_ef_mask;
device_result far* result_ptr;

word Write_dev ( device_handle, buffer, size, AST_NOTIFY,
                 task_id, ast_routine[, ast_handle] );
void (far* ast_routine)( ast_blk far* );
word       ast_handle;
```

Description

This function writes data to an I/O channel which was previously opened; the *device_handle* must be a value returned by **Open_dev**. The *buffer* parameter contains the **far** address of the data to be written, and *size* contains the number of data items to write. If the channel was opened in NATIVE_MODE, *size* specifies a number of bits, bytes, or words, depending on the type of the requested data. Otherwise, *size* specifies a number of bytes.

The *notify_code* specifies the method used to notify the calling task that the operation has completed; its value may be WAIT, EVENT_NOTIFY, or AST_NOTIFY. When WAIT is used, there are no *nowait options*, and the return from **Write_dev** is delayed until the operation completes. The other *notify_code* values cause the function to return immediately, allowing the calling task to continue execution.

When EVENT_NOTIFY is used, *local_ef_mask* is a word with one or more bits set; these bits correspond to the local event flags which VTOS will set when the operation completes. The calling task should ensure that the event flag or flags are not already set by using **Reset_ef** to reset them before calling **Write_dev**. When the operation has completed, the structure at *result_ptr* will contain status information. Note that the *result_ptr* parameter must be explicitly cast as a **far** pointer because its type is not specified by the function prototype in **vtos.h**. If the call succeeds, the **ioreturn** member of the structure at *result_ptr* contains the number of data units written, and the **iostatus** member contains SUCCESS; when a failure occurs, **ioreturn** contains the number of characters written at the time when the failure occurred, and **iostatus** contains an error status code. For a discussion of asynchronous I/O using event flags, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

When AST_NOTIFY is used, VTOS posts an asynchronous trap (AST) after the operation completes. The *ast_routine* contains the name of a function to handle the AST. The optional *ast_handle* contains a user-selected tag value for this particular operation, to permit the AST function to identify it, if necessary. When VTOS calls *ast_routine*, it passes the address of an **ast_blk** structure. The *ast_handle* value is in the **handle** member of the **ast_blk**. If the call succeeds, the **arg2** member of the **ast_blk** contains the number of data units transferred, and the **arg1** member contains SUCCESS; when a failure occurs, **arg2** contains IO_FAILED, and **arg1** contains an error status code. For a discussion of asynchronous I/O using AST functions, see chapter 6, *Real-Time Programming*, in the *C Programmer's Toolkit for Series 90 PCMs User's Manual*, GFK-0771.

■ Return Value

In WAIT mode, the function return value contains the number of data items actually written. When an error occurs, the return value will be less than *size*. A status code value is available in the global variable **_VTOS_error**.

In EVENT_NOTIFY and AST_NOTIFY modes, the value returned by the function call is undefined and may be ignored. Separate return and status values are available in **device_result** and **ast_blk** structures, respectively.

For all modes, the return and status variables contain values from this table.

Return Value	Status Value	Completion Status
Equal to *size*	SUCCESS	The specified number of data items was written.
Less than *size*	ABORTED	An EVENT_NOTIFY or AST_NOTIFY call was aborted before the write was completed.
	BAD_HANDLE	An invalid *device_handle* was specified. No data was written.

See Also

Close_dev, Open_dev, Read_dev, Seek_dev

Example

```
#include <vtos.h>

word words_written, task, handle;
word buf[] = { 1, 2, 3, 4, 5, 6, 7, 8 };

task = Get_task_id();

handle = Open_dev( "CPU:%R12", READ_MODE | WRITE_MODE | NATIVE_MODE,
                   WAIT, task );

words_written = Write_dev( handle, buf, sizeof( buf )/sizeof( word ),
                           WAIT, task );
if (_VTOS_error == SUCCESS) {
     /* The number of words is in words_written. */
}
```

This example uses a WAIT mode **Write_dev** request to write eight words to the PLC register table, beginning at %R12.

write_localdata

Usage

```
#include <prgmem.h>

REQSTAT write_localdata ( session_id, program_task_name,
                          subblock_name, begin_addr, end_addr,
                          data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
char far* subblock_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

This function writes the specified data to %L (local subblock) registers in the specified Series 90-70 subblock in the specified main program. This request is valid only for Series 90-70 PLCs. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *program_task_name* pointer must contain the address of a NUL terminated ASCII string holding the name of the control program task that owns the target subblock, and *subblock_name* must point to a NUL terminated ASCII string holding the subblock name. Valid names consist of seven characters or less, not counting the NUL character. The *begin_addr* parameter contains the index where the target data begins, and *end_addr* contains the index where the data ends.

When the function succeeds, the data at *data_buffer_ptr* is copied to the range of %L registers specified by *begin_addr* and *end_addr* in the program subblock specified by *program_task_name* and *subblock_name*.

The data is treated as 16 bit binary integer values. The actual register content, however, may be signed or unsigned integers, floating point values, or text.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOT_FOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC programtask.
INVALID_PARAMETER	REQUEST_ERROR	The *subblock_name* is not the name of a subblock in the specified program, or *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem, read_sysmem_nowait, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

■ Example

```
#include <prgmem.h>

WORD buf[7] = { 1,2,3,4,5,6,7 };
REQSTAT status;

/*
 * To write %L1 through %L7, inclusive in the subblock named
 * ”MYBLOCK” of the program named ”MYPROG”:
 */

status = write_localdata( session_id, ”MYPROG”, ”MYBLOCK”, 1, 7, buf ):

/*
 * To write %L28 only in the subblock named ”SUB1” of the
 * program named ”LOADER”:
 */

status = write_localdata( session_id, ”LOADER”, ”SUB1”, 28, 28, buf );
```

This example uses two WAIT mode requests to write to %L data in the specified PLC program subblocks.

write_localdata_nowait

■ Usage

```
#include <prgmemnw.h>

REQID write_localdata_nowait ( session_id, program_task_name,
                               subblock_name, begin_addr,
                               end_addr, data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
char far* subblock_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

■ Description

See **write_localdata**.

■ Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOT_FOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC program task.
INVALID_PARAMETER	REQUEST_ERROR	The *subblock_name* is not the name of a subblock in the specified program, or *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem, read_sysmem_nowait, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata, write_prgmdata, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

Example

```
#include <prgmemnw.h>

WORD value = 0x1234;
REQID request_id;
REQSTAT status;

request_id = write_localdata_nowait( session_id, ”LOADER”, ”SUB1”,
                                     28, 28, &value );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
        status = reqstatus ( request_id, TRUE );
        /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the %L data was written */
}
```

This example uses a NOWAIT mode request to write to %L data in the specified PLC program subblock.

write_prgmdata

Usage

```
#include <prgmem.h>

REQSTAT write_prgmdata ( session_id, program_task_name,
                         begin_addr, end_addr, data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

This function writes the specified data to %P (program) registers in the specified Series 90-70 program. This request is valid only for Series 90-70 PLCs. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *program_task_name* pointer must contain the address of a NUL terminated ASCII string holding the name of the target program. Valid names consist of seven characters or less, not counting the NUL character. The *begin_addr* parameter contains the index where the target data begins, and *end_addr* contains the index where the data ends.

When the function succeeds, the data at *data_buffer_ptr* is copied to the range of %P registers specified by *begin_addr* and *end_addr* in the program specified by *program_task_name.*

The data is treated as 16 bit binary integer values. The actual register content, however, may be signed or unsigned integers, floating point values, or text.

The following table shows examples of target %L data ranges and their corresponding *begin_addr* and *end_addr* values.

Program Data Range	*begin_addr* Value	*end_addr* Value
%P1 through %p24, inclusive.	1	24
%P39 through %p43, inclusive.	39	43
%P500 only.	500	500

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOT_FOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC program task.
INVALID_PARAMETER	REQUEST_ERROR	The *subblock_name* is not the name of a subblock in the specified program, or *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem, read_sysmem_nowait, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata_nowait, write_sysmem, write_sysmem_nowait

Example

```
#include <prgmem.h>

REQSTAT status;
WORD data[16] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 );

status = write_prgmdata ( sesn_id, "MYPROG", 497, 512, data );
```

This example uses a WAIT mode request to write to %P497 through %P512, inclusive.

write_prgmdata_nowait

Usage

```
#include <prgmemnw.h>

REQID write_prgmdata_nowait ( session_id, program_task_name,
                              begin_addr, end_addr,
                              data_buffer_ptr );
BYTE      session_id;
char far* program_task_name;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

See **write_prgmdata**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
TASK_NAME_NOT_FOUND	REQUEST_ERROR	The *program_task_name* is not the name of a PLC programtask.
INVALID_PARAMETER	REQUEST_ERROR	The *subblock_name* is not the name of a subblock in the specified program, or *end_addr* is less than *begin_addr* or out of range.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem, read_sysmem_nowait, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata, write_sysmem, write_sysmem_nowait

Example

```
#include <prgmemnw.h>

REQID request_id;
REQSTAT status;
WORD data[16] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16 );

request_id = write_prgmdata_nowait( sesn_id, ”MYPROG”, 497, 512, data );

if ( request_id < REQUEST_OK ) {
    status = request_id;
} else {
    do {
       status = reqstatus ( request_id, TRUE );
       /* do something else useful */
    } while ( status == REQUEST_IN_PROGRESS );
}

if ( status != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the %P data was written */
}
```

This example uses a NOWAIT mode request to write to %P497 through %P512, inclusive.

write_sysmem

Usage

```
#include <sysmem.h>

REQSTAT write_sysmem ( session_id, memory_type, begin_addr,
                       end_addr, data_buffer_ptr );
BYTE      session_id;
BYTE      memory_type;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

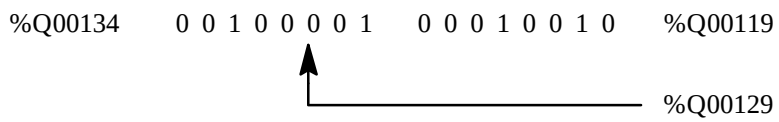
Description

This function writes the specified data to the specified PLC reference (%I, %Q, %R, etc.) in the PLC CPU. The *session_id* must be a value returned by a previous, successful call to **establish_comm_session**. The *memory_type* parameter specifies the target memory type, *begin_addr* contains the index where the target data begins, and *end_addr* contains the index where the data ends. The *end_addr* value must be greater than or equal to *begin_addr*, and must not be larger than the size returned by **get_memtype_sizes** for the specified *memory_type*. The *begin_addr* value must be a value in the range from one (1) to *end_addr*, inclusive.

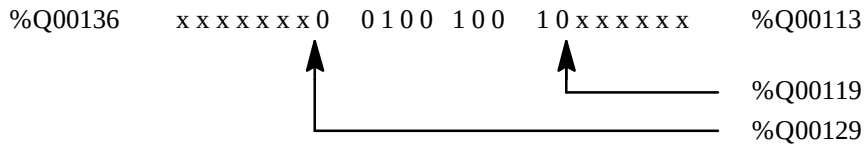
When the function succeeds, the data at *data_buffer_ptr* is copied to the range of PLC references specified by *memory_type*, *begin_addr*, and *end_addr*.

For PLC register data types, %R, %AI, and %AQ, the data at *data_buffer_ptr* is treated as 16 bit binary integer values. The actual register content, however, may be signed or unsigned integers, floating point values, or text.

For the PLC discrete data types, %I, %Q, %M, %T, %S, %SA, %SB, %SC, and %G, the data at *data_buffer_ptr* is expected to be one or more bytes, with the bit specified by *begin_addr* in the least significant bit of the least significant byte. For example, consider the outputs %Q00119 through %Q00129, inclusive. A **write_sysmem** request to send these 11 outputs to the PLC CPU would require the data to be arranged in two bytes as shown here:



Note that the internal PLC format of this data is quite different, as can be seen on the Logicmaster 90 programming software Reference Tables display. The 11 points of the example span three PLC bytes; the ‘‘x’’ values indicate that the original PLC data at that location is unchanged:



The following table shows *memory_type* values from **memtypes.h** which are valid in **write_sysmem** requests.

Reference Type	Access Type	Data Type	*memory_type* Value
%AI	Analog Input Register	WORD	AI_DATA
%AQ	Analog Output Register	WORD	AQ_DATA
%R	Register Memory	WORD	R_DATA
%I	Input Status Table	Discrete	I_STATUS
%Q	Output Status Table	Discrete	Q_STATUS
%T	Temporary Status Table	Discrete	T_STATUS
%M	Internal Status Table	Discrete	M_STATUS
%SA	System A Status Table	Discrete	SA_STATUS
%SB	System B Status Table	Discrete	SB_STATUS
%SC	System C Status Table	Discrete	SC_STATUS
%S	System Status Table	Discrete	S_STATUS
%G	Global Genius Status Table	Discrete	G_STATUS

The Series 90-70 types %GA, %GB, %GC, %GD, and %GE are accessed as subtypes of %G data, as shown in the description of **read_sysmem**.

Return Value

The function returns a REQSTAT value which contains the completion status of the requested operation. When the request succeeds, REQUEST_OK is returned; otherwise, values from this table are returned.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *end_addr* is less than *begin_addr* or out of range.
NULL_SEGSEL_PTR	REQUEST_ERROR	The *memory_type* is invalid.
INVALID_SELECTOR	REQUEST_ERROR	The *memory_type* is not supported for this request.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem, read_sysmem_nowait, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem_nowait

Example

```
#include <sysmem.h>

REQSTAT status;
BYTE sesn_id;
WORD outputs = 0x2112;

status = write_sysmem ( sesn_id, Q_STATUS, 119, 131, &outputs );
```

This example uses a WAIT mode request to send the 11 outputs %Q00119 through %Q00131 shown in the description above.

write_sysmem_nowait

Usage

```
#include <sysmemnw.h>

REQID write_sysmem_nowait ( session_id, memory_type, begin_addr,
                            end_addr, data_buffer_ptr );
BYTE      session_id;
BYTE      memory_type;
WORD      begin_addr;
WORD      end_addr;
void far* data_buffer_ptr;
```

Description

See **write_sysmem**.

Return Value

The function returns a REQID value. When no error is detected in the request, it is sent to the PLC CPU and a value of zero or greater is returned. This value may be used to check the status of the request by calling **reqstatus**. When REQID or the REQSTAT value returned by **reqstatus** is negative, it contains a value from this table.

Most Significant Byte	Least Significant Byte	Error Condition
INVALID_PARAMETER	REQUEST_ERROR	The *end_addr* is less than *begin_addr* or out of range.
NULL_SEGSEL_PTR	REQUEST_ERROR	The *memory_type* is invalid.
INVALID_SELECTOR	REQUEST_ERROR	The *memory_type* is not supported for this request.
DEVICE_NOT_AVAILABLE	NO_COMMUNICATION	Communication has not been established through calls to **api_initialize**, **configure_comm_link**, and **establish_comm_session**.
NO_SMEM_AVAIL	REQUEST_ERROR	An attempt to allocate memory for the request failed.
NO_UMEM_AVAIL	REQUEST_ERROR	There are 256 NOWAIT requests already outstanding.

See Also

establish_comm_session, get_memtype_sizes, get_memtype_sizes_nowait, read_localdata, read_localdata_nowait, read_sysmem, read_sysmem_nowait, read_prgmdata, read_prgmdata_nowait, reqstatus, write_localdata, write_localdata_nowait, write_prgmdata, write_prgmdata_nowait, write_sysmem

Example

```
#include <sysmemnw.h>

REQID reqid1, reqid2;
REQSTAT stat1, stat2;
BYTE sesn_id;
WORD aq_data[4] = { 12345, 12346, 12347, 12348 };
BYTE m_status[2] = { 0x55, 0xaa };

/* Write analog output data %AQ0009 through %AI0012: */
reqid1 = write_sysmem_nowait ( sesn_id, AQ_DATA, 8, 12, aq_data );

if ( reqid1 < REQUEST_OK ) {
    stat1 = reqid1;
} else {
    stat1 = reqstatus ( reqid1, TRUE );
}
/* Write internal contacts %M00037 through %M00045: */
reqid2 = write_sysmem_nowait ( sesn_id, M_STATUS, 37, 45, m_status );

if ( reqid2 < REQUEST_OK ) {
    stat2 = reqid2;
} else {
    stat2 = reqstatus ( reqid2, TRUE );
}
while ( stat1 == REQUEST_IN_PROGRESS || stat2 == REQUEST_IN_PROGRESS ) {
    if ( stat1 == REQUEST_IN_PROGRESS ) {
       stat1 = reqstatus ( reqid1, TRUE );
    }
    if ( stat2 == REQUEST_IN_PROGRESS ) {
       stat2 = reqstatus ( reqid2, TRUE );
    }
}
if ( stat1 != REQUEST_OK || stat2 != REQUEST_OK ) {
    /* investigate the error */
} else {
    /* the new analog input data is available */
}
```

This example uses a NOWAIT mode request to send the four analog outputs %AQ0009 through %AQ0012 and the 8 internal coils %M00037 through %M00045.

S

T

U

V

W