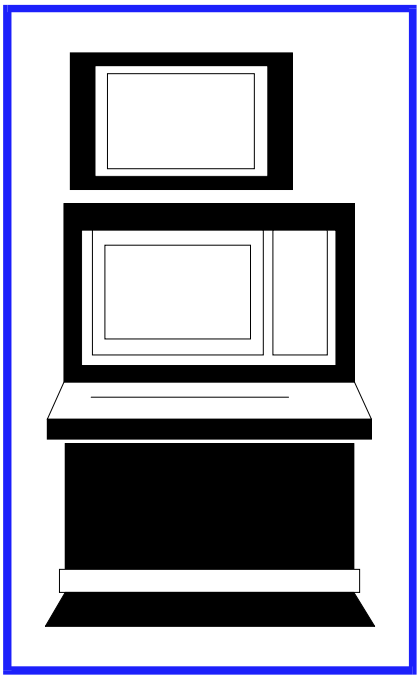
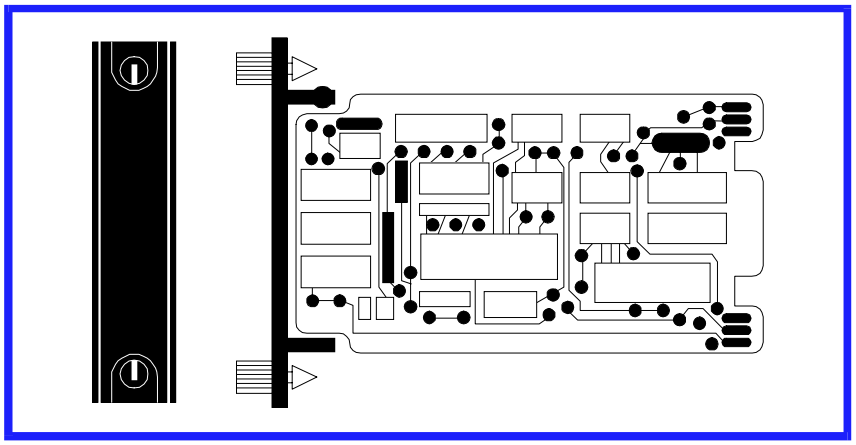
Bailey® infi 90®

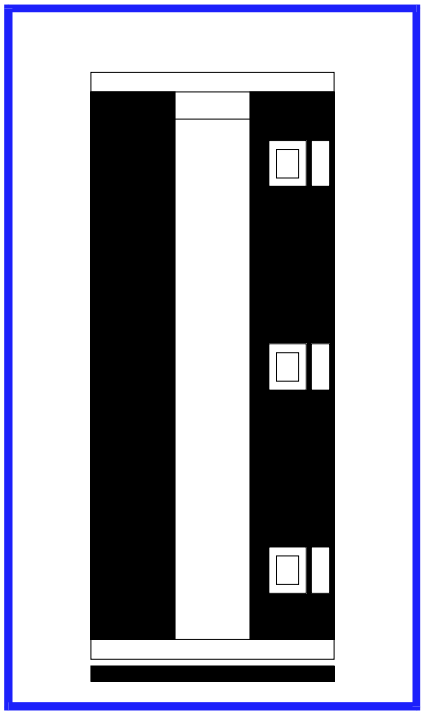
LSIMB01

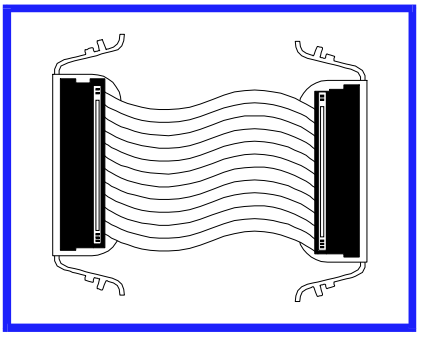
Instruction

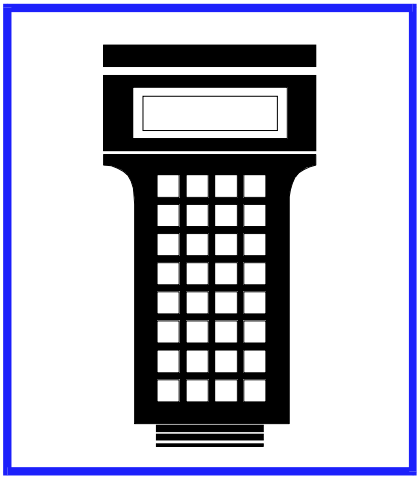
Simulation Network API
(Release 2.0)

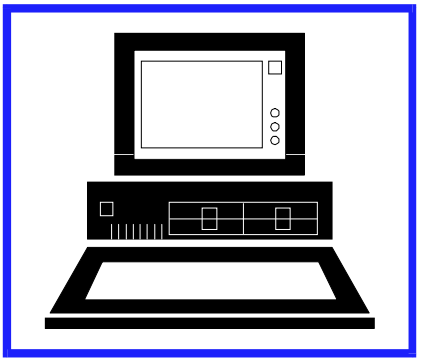












Bailey Process Control and Automation Solutions from Elsag Bailey Group

WARNING notices as used in this instruction apply to hazards or unsafe practices that could result in personal injury or death.

CAUTION notices apply to hazards or unsafe practices that could result in property damage.

NOTES highlight procedures and contain information that assists the operator in understanding the information contained in this instruction.

WARNING

INSTRUCTION MANUALS

DO NOT INSTALL, MAINTAIN, OR OPERATE THIS EQUIPMENT WITHOUT READING, UNDERSTANDING, AND FOLLOWING THE PROPER **Elsag Bailey** INSTRUCTIONS AND MANUALS; OTHERWISE, INJURY OR DAMAGE MAY RESULT.

RADIO FREQUENCY INTERFERENCE

MOST ELECTRONIC EQUIPMENT IS INFLUENCED BY RADIO FREQUENCY INTERFERENCE (RFI). CAUTION SHOULD BE EXERCISED WITH REGARD TO THE USE OF PORTABLE COMMUNICATIONS EQUIPMENT IN THE AREA AROUND SUCH EQUIPMENT. PRUDENT PRACTICE DICTATES THAT SIGNS SHOULD BE POSTED IN THE VICINITY OF THE EQUIPMENT CAUTIONING AGAINST THE USE OF PORTABLE COMMUNICATIONS EQUIPMENT.

POSSIBLE PROCESS UPSETS

MAINTENANCE MUST BE PERFORMED ONLY BY QUALIFIED PERSONNEL AND ONLY AFTER SECURING EQUIPMENT CONTROLLED BY THIS PRODUCT. ADJUSTING OR REMOVING THIS PRODUCT WHILE IT IS IN THE SYSTEM MAY UPSET THE PROCESS BEING CONTROLLED. SOME PROCESS UPSETS MAY CAUSE INJURY OR DAMAGE.

NOTICE

The information contained in this document is subject to change without notice.

Elsag Bailey, its affiliates, employees, and agents, and the authors and contributors to this publication specifically disclaim all liabilities and warranties, express and implied (including warranties of merchantability and fitness for a particular purpose), for the accuracy, currency, completeness, and/or reliability of the information contained herein and/or for the fitness for any particular use and/or for the performance of any material and/or equipment selected in whole or part with the user of/or in reliance upon information contained herein. Selection of materials and/or equipment is at the sole risk of the user of this publication.

Preface

This instruction provides information on software release 2.0 of the simulation network application programming interface. The simulation network application program interface (API) software provides a simulation network communication system and an application program interface to that communication system. Using this software, the INFI 90® OPEN Strategic Process Management System, LMUSM02/04 Universal Simulation Modules, and process simulation software can be interconnected to create a realistic and cost effective simulator. User tasks written with the C programming language can be run on the HP-UX® operating system (the operating system used by this software).

Software release 2.0 of the simulation network API includes the following enhancements:

- Support for the following modules:

  IMRIO02 Remote Input/Output Module.
  IMSED01 and IMSET01 Sequence of Events Modules.
  IMFBS01 Field Bus Slave Module.

- Save and restore large snapshots.

- Improved diagnostic messages: The status of internal USM variables such as the USM working mode, communications statistics, and connection information are now available to the simulation manager.

- Improved restart robustness.

- Additional simulation API function calls:
  sn90_snapshot_delete()
  sn90_snapshot_delete_ack()
  sn90_get_snapshot_delete()

List of Effective Pages

Total number of pages in this instruction is 159, consisting of the following:

Page No.	Change Date
Preface	Original
List of Effective Pages	Original
iii through viii	Original
1-1 through 1-7	Original
2-1 through 2-8	Original
3-1 through 3-5	Original
4-1 through 4-19	Original
5-1 through 5-74	Original
6-1 through 6-18	Original
7-1	Original
8-1 through 8-3	Original
A-1 through A-2	Original
B-1 through B-10	Original
C-1 through C-2	Original
Index-1 through Index-2	Original

When an update is received, insert the latest changed pages and dispose of the superseded pages.

NOTE: On an update page, the changed text or table is indicated by a vertical bar in the outer margin of the page adjacent to the changed area. A changed figure is indicated by a vertical bar in the outer margin next to the figure caption. The date the update was prepared will appear beside the page number.

Table of Contents

Table of Contents (continued)

Table of Contents (continued)

Table of Contents (continued)

List of Figures

List of Tables

Trademarks and Registrations

Registrations and trademarks used in this document include:

® Hewlett-Packard	Registered trademark of Hewlett Packard Company.
® HP	Registered trademark of Hewlett Packard Company.
® HP-UX	Registered trademark of Hewlett Packard Company.
® INFI 90	Registered trademark of Elsag Bailey Process Automation.

SECTION 1 - INTRODUCTION

OVERVIEW

The INFI 90 OPEN simulation system incorporates industry standard process simulator features into the INFI 90 OPEN Strategic Process Management System. The simulation network system includes the simulation network application program interface (API) software and the universal simulation module (USM).

The unique feature of a simulator incorporating the Bailey simulation network is that an actual control system rather than an emulated control system controls the process simulation. This is an important feature when training operators, checking out the control system, or optimizing the controls.

This instruction documents the simulation network API software. The API software supports a simulation environment by providing a communication system for simulations and an application program interface to the communications system. The simulation network API software facilitates communications between a simulation application task and the INFI 90 OPEN system.

Figure 1-1 shows an overview of a typical simulator system with respect to communications. Components of a typical simulator system include:

Simulation network - provides the functions necessary to support the simulation applications.

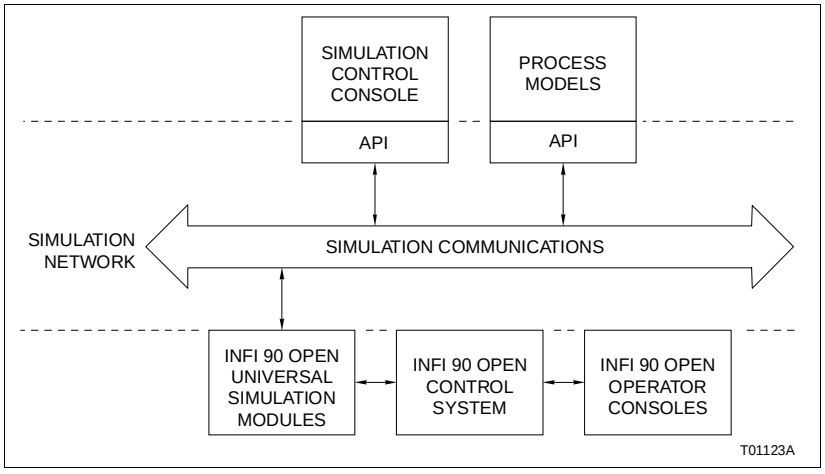


Figure 1-1. Typical Simulation System

Application program interface - used to access the INFI 90 OPEN system, via the simulation network communication system, from the simulation application.

INFI 90 OPEN system - standard INFI 90 OPEN system with the multi-function processor (MFP) modules updated for use with the simulation network. The field I/O to the MFP modules that normally is handled by I/O modules is replaced by I/O through USM modules. The USM modules provide an interface between the simulation network (ethernet) and the MFP modules via the I/O expander bus.

Process models - set of equations that define the dynamics of a piece of process equipment. This may be a third-party item.

Simulation control application - interface to the process models and the simulation manager for controlling the simulator. An instructor console is an example of this type of application. This may be a third-party item.

Universal simulation modules - module that serves as a link between the simulation network (ethernet) and the I/O expander bus. The USM module simulates the presence and behavior of INFI 90 OPEN I/O modules. It is configured by the simulation network API software when the simulation is started. The redirection of I/O from the MFP module to the USM module does not require any changes to the MFP module configuration logic.

The simulation network communication system provides the functions necessary to support the basic simulation functions. These functions include communications management, freeze/run, snapshot save/restore, message delivery, and logging.

Refer to the appropriate document listed in ***REFERENCE DOCUMENTS*** for more information on MFP and USM modules. Refer to the third-party documentation for information on their respective items. The remainder of this instruction focuses on the simulation network API software.

Simulation Network Communications

An ethernet (TCP/IP) network provides the communications link between the various processors in the simulator. The processors connected to the simulation network include USM modules and computers running process simulations. The simulation network tasks could include the following:

- USM/MFP module pairs.
- Process models (Bailey or third-party).
- Simulation control application (Bailey or third-party).

The simulation control application and process models can be on one computer or distributed on several computers.

Simulation Network API

The purpose of the simulation network API software is to provide a programming interface to the simulation network. It also provides a means for third-party programs to interface with the Bailey simulation network. The simulation network connects the simulation applications to the USM modules using ethernet. The simulation network connects simulation tasks executing on different computers or tasks executing on the same computer. The simulation network API software handles the details of the task-to-task communications on the simulation network allowing programmers to concentrate on their application.

The simulation network API software contains a set of functions that a programmer uses to execute the functions required in a process simulator. It is provided in a C programming language format on the HP-UX operating system.

INTENDED USER

This instruction provides the information necessary to install and use the simulation network API software. It is intended for C programmers that understand the design of simulation application software. The programmer must also be familiar with the INFI 90 OPEN Strategic Process Management System.

REQUIREMENTS

The simulation network API software runs on a Hewlett-Packard® work station. The work station must be equipped with:

- HP-UX version 9.0 or later operating system.
- HP® C or C++ programming language compatible with the operating system.
- 14 megabytes of free hard disk space.
- Digital audio tape (DAT) drive.

The INFI 90 OPEN system must utilize IMMFP01, IMMFP02 and IMMFP03 Multi-Function Processor modules with firmware revision F.0 or later. Revision G.0 firmware is required if remotely mounted I/O modules are to be simulated.

FEATURES

Some of the advantages of using a simulator that incorporates the simulation network API software are:

- Control logic developed for a real application is downloaded unchanged into the simulator. There is no need to translate control logic into another form for execution on a computer.
- More realistic and useful training can be provided because the simulator is using real control logic and control system configurations.
- Reduced maintenance costs of the simulator because it uses actual plant control logic, hardware, and system configurations. Changes to the actual control system could easily be transferred to the simulator.
- Testing and optimizing of actual control logic can be done on the simulator since it can duplicate actual plant performance to a high degree of fidelity.

The simulation network API software provides a means by which an application can use the following INFI 90 OPEN simulation features:

- Point I/O with MFP modules.
- Freeze/run capabilities.
- Slow and fast execution rates.
- Snapshot save and restore of the MFP module state.
- Emulation of INFI 90 OPEN module failures.

INSTRUCTION CONTENT

This instruction contains eight sections and three appendices. It also includes a Table of Contents, List of Figures, List of Tables, and Index giving several options to locate specific information quickly. Appendices supplement information presented in the individual sections. The sections that make up this instruction include:

Introduction — Provides an overview of the simulation network API software and this instruction.

Description and Operation — Describes the capabilities of the simulation network API software and how they are implemented.

Installation — Explains how to load the simulation network API software and the resulting directory structure.

Function Library Overview — An overview of the functions provided in the API function library.

Function Library — Provides detailed information about each API function.

Operating Procedures — Describes how to configure, start, and run a simulation. Explains how to compile and link applications.

Error Messages and Recovery — Lists the possible error messages and corrective actions required.

Message Handling — Describes notification messages, how they are handled, and the appropriate response to them.

HOW TO USE THIS INSTRUCTION

Read this instruction through in sequence before attempting to install or operate the simulation network API software. It is important to become familiar with the entire contents of the instruction prior to performing any procedures to insure maximum use of all available functions.

This instruction limits the information presented in each section to only specific items required to complete the desired task. The organization enables finding specific information quickly, and permits using this instruction as a reference after becoming fully familiar with the software.

Read the notes in text. Notes provide:

- Additional information.
- Information that should be considered before performing a certain operation.

DOCUMENT CONVENTIONS

This document uses standard text conventions throughout to represent keys, user data inputs and display items:

KEY — Identifies a keyboard key.

Example: Press **ENTER**.

Display item — Any item that displays on the screen appears as italic text in this document.

Examples: *Save Setup File* (menu selection)

Installation Complete (message)

Tag name or index number (prompt)

File name — Any file names and file extensions appear as bold-italic text.

Example: ***taskio.C***

This document uses a specific set of text conventions for commands:

BOLD Identifies any part of a command line that is ***not*** optional or variable, and must be entered exactly as shown.

italic Identifies a variable parameter in a command line.

[ ] Identifies a parameter that is optional.

Example: **/sn90/bin/sn90sm** *simul_ID* **[-d** *staticDir***]**

GLOSSARY OF TERMS AND ABBREVIATIONS

Table 1-1 is a glossary of terms and abbreviations used in this instruction.

Table 1-1. Glossary of Terms and Abbreviations

Term	Definition
API Services	A set of C language functions that control the execution of the API functions.
I/O Expander Bus	Parallel communication bus between the control and I/O modules.
MFP	Multi-function processor module. A multiple loop controller with data acquisition and information processing capabilities.
OIS	Operator interface console. Integrated operator console with data acquisition and reporting capabilities. It provides a digital access into the process for flexible control and monitoring.
Process Model	A set of equations that define the dynamics of a piece of process equipment.
Simulation Control Application	A task (typically running on a separate platform) that is an interface used to manipulate a simulation.
Simulation Manager	A task that coordinates the tasks and resources required for a simulation.
Simulation Network API	The programming interface that allows applications to communicate with the INFI 90 OPEN system and other simulation applications over the Bailey simulation network or by interprocess communications.
Simulation Network Services	A set of functions that provide the ability to send and receive data over the simulation network.
Task Manager	A local task (running on each simulation computer) that collects local information and distributes this information to task managers running on other simulation computers. It collects corresponding information about tasks from other task managers in the network.
USM	Universal simulation module. A module that allows field I/O to be replaced with I/O to a computer executing process models on an ethernet system.

REFERENCE DOCUMENTS

This instruction provides information about the simulation network API software only. Table 1-2 lists additional documents that relate to the hardware.

Table 1-2. Reference Documents

Number	Document
I-E96-201	Multi-Function Processor Module (IMMFP01)
I-E96-202	Multi-Function Processor Module (IMMFP02)
I-E96-203	Multi-Function Processor Module (IMMFP03)
I-E96-323	Universal Simulation Module (LMUSM01/02/04)
P-E96-752-001A	Simulation Network Application

SECTION 2 - DESCRIPTION AND OPERATION

INTRODUCTION

This section describes the simulation network software, its parts, and how they function. Due to the complexity and infinite number of possible implementations of the software, examples are used to illustrate software functionality.

EXAMPLE SIMULATORS

There are many possible variations (multiple simulation computers, multiple USM modules, etc.) for a process simulator. Figures 2-1 and 2-2 demonstrate two possible simulator setups. The following discussions of how each simulator works will provide a general understanding of simulators. A detailed explanation of the simulation network API software and its component parts is provided later in this section.

Example Simulator One

The simulator shown in Figure 2-1 consists of a computer running the simulation network API software and three USM/MFP module pairs. The modules are connected to an INFI 90 OPEN system that contains an operator interface station (OIS) console. This simulator has only one process simulation task.

Powering up the simulator should cause all tasks (including the simulation manager and task manager) to be loaded, opened, and registered. When the start-up procedures are complete, the simulation computer provides access for the simulation control application. The simulation control application and process model can be third party software packages. The simulation control application is used to instruct the simulation manager to perform operations. Use the simulation control application to start the simulation and process models. The USM modules receive values from the process model and pass them to the MFP modules and the INFI 90 OPEN system. From the OIS console, it will seem as though a real process is being controlled.

During the execution of a simulation, it may be desirable to save a snapshot of the simulator state for later use or reference. The simulation control application can be used to request a snapshot save or it can automatically save snapshots at a certain time interval. To load a saved condition, use the simulation control application to restore the simulator to the saved condition. The restored snapshot can be used as the initial condition for the start of a simulation session. INFI 90 OPEN malfunctions can be simulated at any time using the

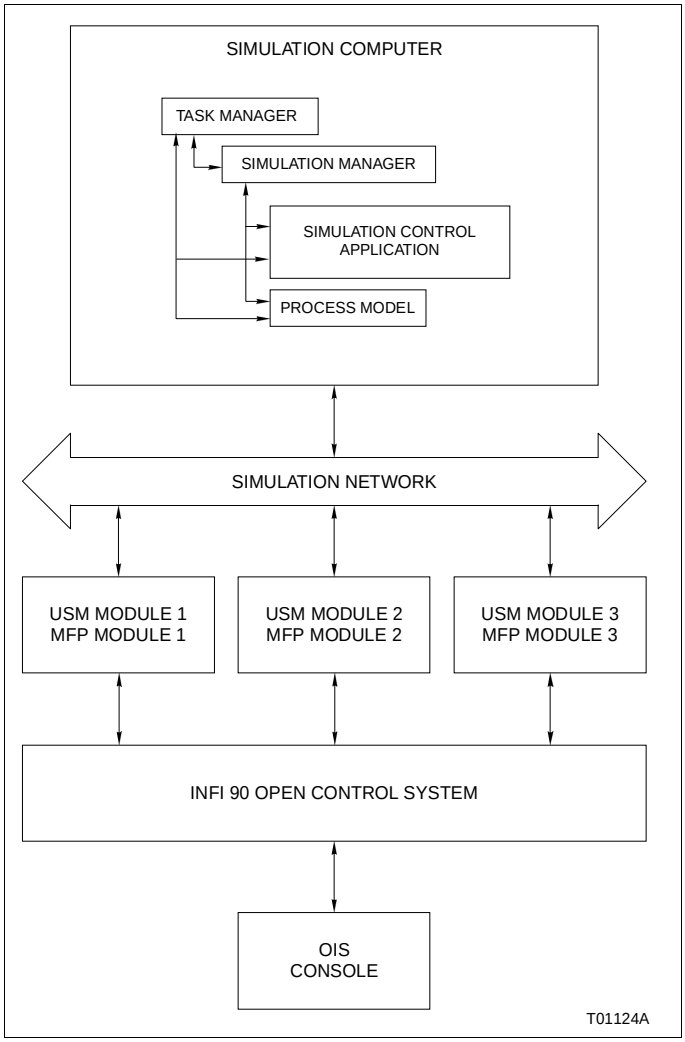


Figure 2-1. Example Simulator Setup One

simulation control application. They can also be programmed into the simulation control application for automatic activation.

Use the simulation control application to freeze (pause) or end the simulation when the simulation session is over.

Example Simulator Two

The simulator shown in Figure 2-2 consists of two computers running the simulation network API software and ten USM/MFP module pairs. The modules are connected to an INFI 90 OPEN system that contains an OIS console. This simulator has two process simulation tasks.

This simulator functions the same as example simulator one with two exceptions. Two process models, each controlled by five USM/MFP module pairs, are used to emulate a more

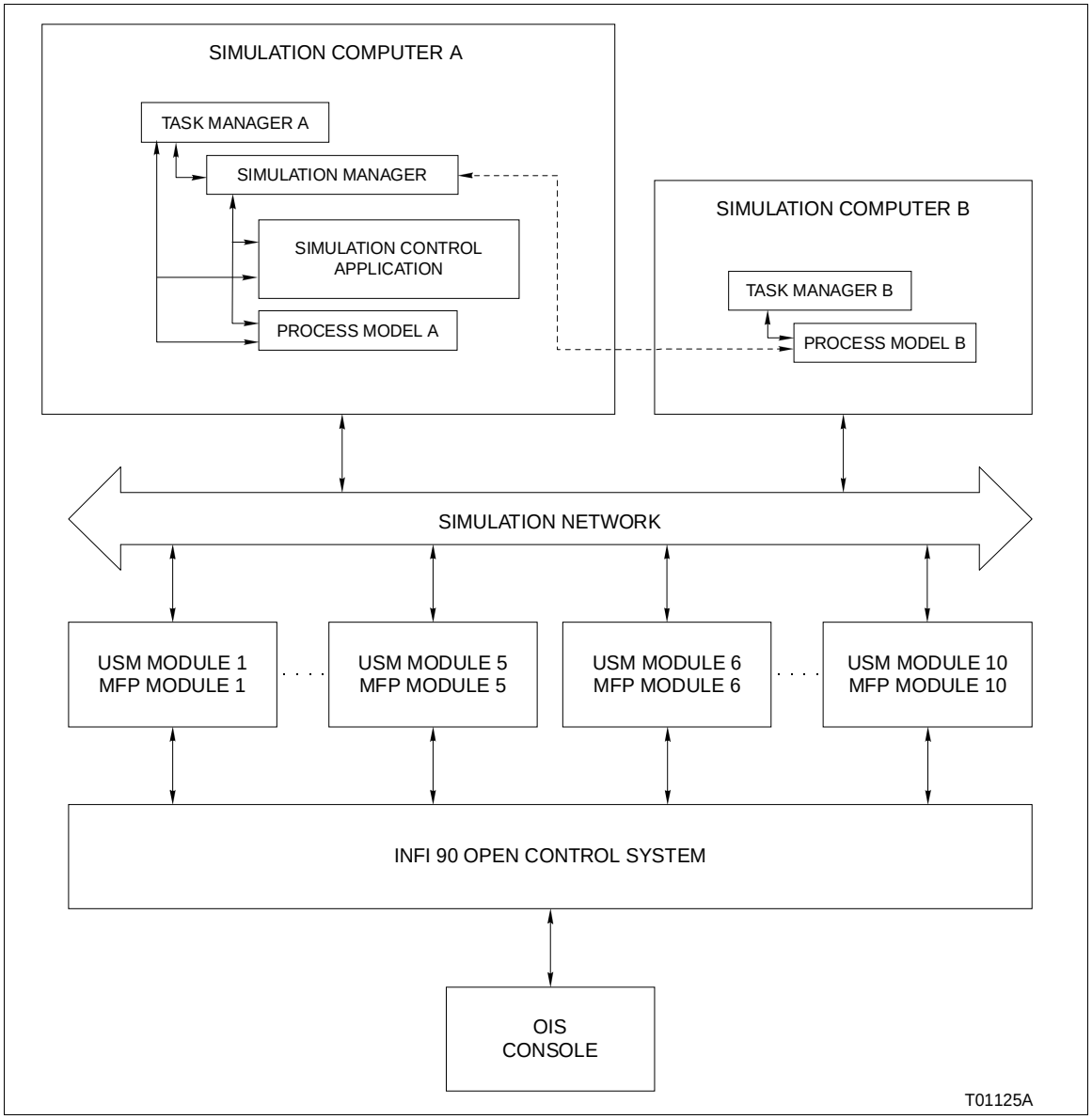


Figure 2-2. Example Simulator Setup Two

detailed and complex control system. The second process model and a task manager are loaded into a separate computer. This configuration is used when the two process models do not fit in the same computer or the computer cannot satisfactorily execute both process models at the same time. Notice that there is only one simulation manager task when distributing the process models on two computers.

SIMULATION NETWORK

The simulation network API software provides a set of software services that manage the simulation network. Figure 2-3 shows a diagram of the simulation network.

Components of the simulation network software are:

- Task manager task.
- Simulation manager task.
- API services.
- Simulation network services.

Typical usage of an operator training simulator would consist of an instructor starting up all tasks used in the simulation. The process models of the simulation send values to the USM/MFP module pairs. Trainees sitting at an operator console would oversee the operation of the process and process control system. At some point an instructor might introduce malfunctions into the process and then assess the corrective actions taken by the trainees. These malfunctions can be scheduled to be activated at a specified time or activated in real-time from a simulation control application. To evaluate trainees' performance, a snapshot of the system can be saved. This snapshot can be used to return the simulation to the saved condition. Another usage of the simulation network would consist of engineers testing new control configurations in an effort to optimize the control system.

The remainder of this section describes the components of the simulation network API software and how they function. An explanation of the simulation API as a whole is also contained in this section.

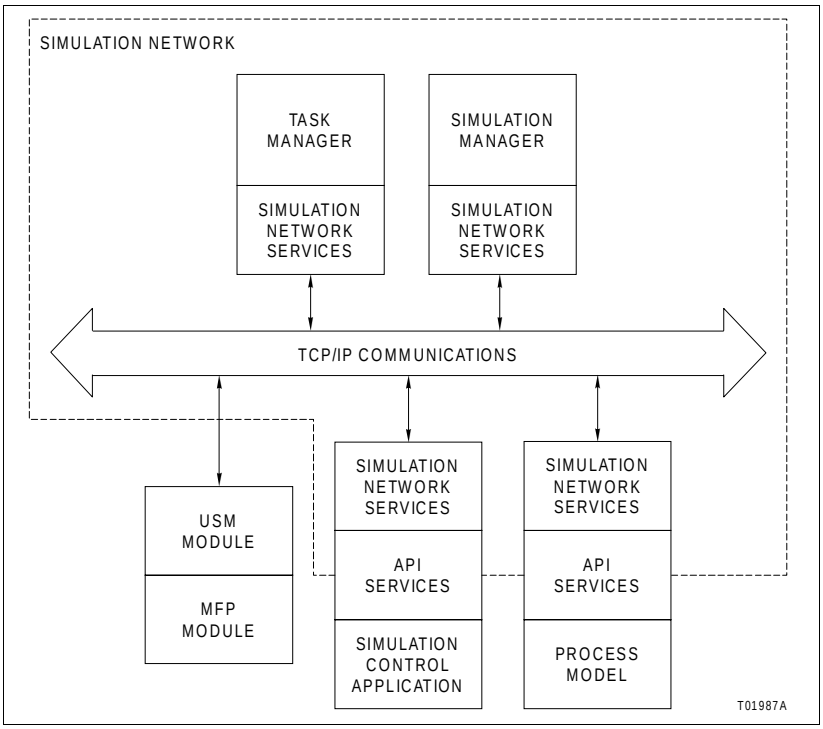


Figure 2-3. Simulation Network Logic Architecture Diagram

Task Manager

The task manager acts as a task location broker for simulations. By being the location broker, the task manager allows client tasks to resolve the location of other pertinent tasks at runtime. The task manager exists for the primary purpose of providing a known address on each computer on the simulation network. This provides a task that all other tasks on that computer can communicate with and thus share their task ID, simulation ID, and network address.

To handle a distributed simulation, the task managers running on different computers exchange information. The simulation network supports multiple simulation tasks (e.g., process models) that can be executing on different computers on the simulation network. A simulation task can be moved from one computer to another without changing configuration information. The simulation network supports the sharing of the simulation ethernet and simulation computers with multiple simulators.

The task manager monitors the existence of simulation tasks on its computer and announces the addition or deletion of a task. This allows a new task to be detected and automatically added to the simulation.

Each simulation task registers with the task manager at start-up using the **sn90_open()** API function. This provides the task manager with information about that task. Also, the API services of the simulation task acquire information about other simulation managers executing on the simulation network.

The task manager must be executing prior to the start-up of any simulation task on that computer.

Simulation Manager

The simulation manager coordinates the multiple tasks that are part of the simulator, such as process simulations and control system modules. It also keeps track of the tasks that are in communication with the simulation, such as a simulation control application.

The simulation network can support several simulation tasks running at the same time. Each simulator has a simulation manager that manages activities relevant to a specific simulation environment. Handling the commands from any simulation network task and distributing the command to other simulation network tasks is one such activity. Managing the collection of snapshots is another major function of the simulation manager.

The functions of the simulation manager include:

- Communication initialization.
- Simulation task status tracking.
- Command logging.
- Snapshot save/restore of INFI 90 OPEN data.

The simulation manager:

- Provides point name cross referencing to determine to which other tasks a client task needs to communicate. Client tasks communicate directly to USM modules or other tasks.

- Provides centralized command routing. All commands are received by the simulation manager and forwarded to the appropriate task.

- Provides centralized simulation network command logging.

- Provides collection, storage, and restoring of snapshot data for USM and MFP modules. The client task needs no knowledge of how snapshot works other than providing the directory and snapshot ID.

- Forwards USM initialization information received from a task to the appropriate USM module.

- Stores the status of each task in the simulator and makes it available on request.

All communication from the client task to the simulation manager is handled through simulation network API functions. The simulation manager takes a point name from a client task and searches all other task point lists and USM point lists for the same point name. A match indicates the two tasks need to communicate if they have corresponding I/O directions (e.g., one sends and one receives). The same point can be sent to more than one reception task.

The simulation manager gathers points into I/O groups. Points that have the same source task, destination task, and I/O interval are put into the same I/O group. Tasks exchange point values as defined by the I/O group. The user assigns a point an I/O direction (send or receive) and an I/O interval using the establish point functions. The simulation manager then assigns the point to an I/O group.

The task manager must be executing before starting the simulation manager. The simulation manager must be executing before starting any other tasks in the simulator.

API Services

Each simulation task has the API services and simulation network services software linked into it. The API services is the set of library routines that implement the API functions. API services communicate with the simulation manager, local task manager, and other tasks via the network services.

The API services perform the following operations:

- Provide the communications functions necessary to interface with the simulation network services.
- Implement an API request to register the task with the task manager so that its address is known to other tasks.
- Request task status information from the simulation manager to determine the status of each task in the simulation.
- Receive API point establish requests to initialize its local point table and make this information available to the simulation manager. The simulation manager uses this information to determine connection requirements.
- Receive API USM configuration information that is passed on to the simulation manager for the purpose of initializing the USM modules.
- Make connections to other tasks that it must exchange information with.
- Receive requests in the form of API function calls and convert them into the appropriate transaction message to the appropriate task. A simulation network services function is called to send the message.
- Receive command messages and acknowledgments, queue the data, unpack the message, and provide it to the client task in the form of an API function argument.
- Build and send I/O values messages when the **sn90_update_values()** function is called ***and*** the I/O interval has been exceeded for the point group. It unpacks and copies values into the appropriate memory location from received I/O values messages.

Simulation Network Services

The simulation network services are a set of functions that are linked into and used by the API services, task manager, and simulation manager. The simulation network services are accessed through the simulation API.

The simulation network services provide the low level functions that are specific to communications connections. These include creating a TCP/IP socket, binding an address to it, connecting the socket to another task, writing a message buffer to the socket, and reading a message buffer from the socket.

Simulation API

The simulation API provides a programming interface in C language by which application tasks can interface with the simulation network. The simulation API is designed to simplify the implementation of an application on the simulation network by removing the burden of handling the details and responsibility of the communications from the application programmer. The simulation network provides access to the USM modules and to other tasks.

Examples of simulation tasks include process simulations and simulation control applications.

PERFORMANCE DATA

Performance of the simulation network is dependent upon the USM module performance and the hardware limitations of the computer running the simulation network software. The USM modules are capable of:

- Connecting up to ten tasks (i.e., process model tasks).
- Supporting up to 20 I/O groups.
- Supporting up to 1,024 local I/O points and up to 1,008 remote I/O points.
- Supporting up to 64 locally mounted I/O modules and up to 63 remotely mounted I/O modules.

The only limitations on the number of tasks (applications, USM modules, etc.) the system will support are the available memory and processor speed of the computer running the simulation software and the ethernet bandwidth.

SECTION 3 - INSTALLATION

INTRODUCTION

This section provides instructions on how to install the simulation network API software.

FULL INSTALLATION

Several separate operations must be performed to completely install the simulation network API software:

- Copying the simulation files from the DAT tape to the appropriate directory on the hard disk drive.
- Using the ***sn90inst*** script file to install the simulation network API software.
- Using the ***c++inst*** script file to install the HP C++ linker software (if a version 3.0 or later C++ compiler is not already installed).

NOTE: Super-user access privileges are required to execute the ***sn90inst*** and ***c++inst*** script files.

Installing Simulation Files

The first operation of the installation process is to extract the simulation files from the DAT tape to the /temp directory on the hard disk drive. To install the files:

NOTES:
1. Create the /sn90 and /temp directories before starting the installation. These directory names are not mandatory and any acceptable directory name can be used.

2. The following instructions assume that directories named /temp and /sn90 are being used. If other names have been chosen, substitute those names where /temp and /sn90 appear.

1. Change to the /temp directory by typing:

   cd /temp ENTER

2. Transfer the files from the tape to the /temp directory by typing:

   tar xf /dev/rmt/0m ENTER

The effect of this command is to create directories named sn90 and c++linker under the /temp directory. Directory structures

under those directories that match the required structures are also created, and files are copied from the tape to their correct directories within this structure. The ***sn90inst*** and ***c++inst*** procedures will copy these files to their final destinations.

Using sn90inst Script File

The second operation in the installation process is to install the simulation network API software. The software consists of the API library, include, executable, and example program files. To install the software:

1. Log in as super-user.

2. Change to the /temp directory. Type:

   cd /temp ENTER

3. Type:

   ./sn90inst ENTER

A listing of ***/etc/services*** file entries that are required by the system will appear.

4. At the *Do you wish to append these to your /etc/services file (y/n)?* prompt, type:

   y ENTER

   NOTE: The existing ***/etc/services*** file is copied to the ***/etc/services.old*** file.

5. At the *(/sn90):* prompt, specify the directory that will contain the simulation network API software files by typing:

   /sn90 ENTER

6. The installation procedure will check whether a simulation package already exists. If one is found, you will be asked whether the existing package should be overwritten or the script should exit. At the *Do you want to Exit or Overwrite the current package? Enter 'e' for exit or 'o' for overwrite (e/o):* prompt, type:

   o ENTER

7. At the *This is your last chance to stop this installation. Do you want to continue (y/n)?* prompt, type:

   y ENTER

The /temp/sn90 directory and its subdirectories and files may now be deleted.

Installation of the simulation network API software files is complete when the *Installation complete* message appears.

Using c++inst Script File

If the HP-UX computer to which the simulation network software is being installed does not already have a C++ compiler installed, a third installation operation is required. This operation consists of installing the required components of the HP C++ linker that allow C compiled applications to link with the simulation network API library. To install the C++ linker:

1. Log in as super-user.

2. Make a backup copy of the /usr/lib and /usr/bin directories and their subdirectories.

3. Change to the /temp directory. Type:

   cd /temp ENTER

4. Type:

   ./c++inst ENTER

5. At the *Do you wish to proceed with the installation (y/n)?* prompt, type:

   y ENTER

6. At the *Have you made a backup copy of these system directories (y/n)?* prompt, type:

   y ENTER

7. At the *Do you wish to check for existing linker files (y/n)?* prompt, type:

   y ENTER

A listing of any existing C++ linker files will be displayed.

8. At the *Do you wish to proceed with the installation (y/n)?* prompt, type:

   y ENTER

   NOTE: Answering **y** to the *Do you wish to proceed with the installation (y/n)?* prompt will save any existing linker files prior to installing the C++ linker.

9. At the *Are you sure you want to do this (y/n)?* prompt, type:

   y ENTER

Installation of the C++ linker files is complete when the *Installation complete* message appears.

This procedure will have created subdirectories named sn90old under various subdirectories of /usr/bin and /usr/lib. These subdirectories will be needed if the new C++ linker needs to be removed and previous copies of its files need to be restored.

Uninstalling the C++ Linker

To remove an accidently installed C++ linker and restore the previous C++ linker:

1. Log in as super-user.

2. Type:

   /temp/c++uninst ENTER

3. At the *Do you wish to proceed with the uninstallation (y/n)?* prompt, type:

   y ENTER

Uninstallation of the C++ linker files is complete when the *Finished* message appears.

This procedure will have removed all of the sn90old subdirectories that were created during the execution of **c++inst**.

Installing Hardware Key

Install the serial port hardware key on one of the RS-232-C serial ports of the computer that will be running the simulation manager task. Use the DB-9 to DB-25 cable provided with the hardware key. When starting the simulation manager, specify the appropriate port in the command line, refer to ***SIMULATION MANAGER COMMAND LINE*** in Section 6.

DIRECTORY STRUCTURE

Table 3-1 lists the directories and files created during the installation of the simulation network API software. It assumes that the default directory names /sn90 and /temp were used.

NOTE: The /temp directory and its subdirectories can be deleted after the installation is complete and has been verified.

Table 3-1. Simulation Network API Directory Structure

Directory	Contents
/sn90	Miscellaneous files.
/sn90/bin	Executable files ***sn90sm***, ***sn90tm,*** etc.
/sn90/examples	Example source code files.
/sn90/include	***sn90_api.h*** include file.
/sn90/lib	API library file.
/temp	Installation script files and subdirectories.
/temp/c++linker	Subdirectories bin and lib.
/temp/c++linker/bin	Linker executable files.
/temp/c++linker/lib	Linker library files.

RUNTIME ONLY INSTALLATION

To install only the files necessary for runtime operation on a computer other than the installation computer:

1. Copy the ***sn90tm*** executable file to every computer that will be executing simulation network tasks.

2. Copy the ***sn90sm*** executable file to the computer that will be executing the simulation manager.

3. Copy the ***sm.cfg*** simulation manager configuration file to the computer that will be executing the simulation manager.

SECTION 4 - FUNCTION LIBRARY OVERVIEW

INTRODUCTION

The application program interface (API) functions are organized into ten groups:

- Task control.
- Simulation control.
- Point and module initialization.
- Messages.
- Point I/O values.
- INFI 90 OPEN malfunction simulation.
- Snapshot data.
- Simulation network querying.
- Simulation time management.
- Log file.

This section explains the functions in each group and when to use them. Refer to Section 5 for detailed information about each function.

REMOTE I/O MODULE SIMULATION

Version 2.0 of the Simulation Network API features the ability to simulate remotely mounted I/O modules that communicate with MFP modules through Remote I/O Slave Modules (IMRIO02).

I/O modules mounted in the same PCU cabinet with the MFP module that is using or supplying their data are identified with a single value, the slave address. Remotely mounted I/O modules must be identified by three values:

- The I/O module address of the IMRIO02 module that is acting as the Remote Master Processor (RMP). This IMRIO02 module is mounted in the same PCU cabinet as the MFP module. The address of the RMP module will be found in Specification 2 of the associated Remote I/O Interface (function code 146) block.

- The remote I/O module processor serial link address that the RMP module is using to communicate with the IMRIO02 module that is acting as the Remote Slave Processor (RSP). This address will be found in Specification 2 of the associated Remote I/O Definition (function code 147) block.

- The I/O module address of the target I/O module in the remote PCU cabinet.

To support this addressing scheme, the following structure definition has been added to the file ***sn90_api.h***:

```
/*
** Slave Address structure to support remote I/O
*/
struct SlaveAddress
{
        short localAddress;
        short remoteNode;    /* -1 in this field implies local slave  */
        short remoteAddress;
};
```

Valid values for all three fields are 0-63.

All API functions that require a I/O module address have been given corresponding functions that take a SlaveAddress structure instead.

When developing the functions to support remote I/O module simulation, care was taken not to require the user to know while the simulation is developed whether a given I/O module is locally or remotely mounted. To support this capability, all functions supporting remote I/O check the remoteNode field of the SlaveAddress structure. If that field contains -1, the I/O module is assumed to be locally mounted, with the I/O module's address contained in the localAddress field. Using this capability, it is possible to develop a complete simulation incorporating both local and remote I/O modules (or local I/O modules exclusively) using only the remote I/O module versions of the API functions.

NOTES:

1. Some log file messages that reported I/O module addresses now report SlaveAddress structures. These structures appear in the form A/B/C, where A is the RMP module's address, B is the serial link address, and C is the I/O module's address in the remote PCU cabinet. If an I/O module is locally mounted, the address will appear in the form A/-1/0, where A is the I/O module's address.

2. MFP module firmware revision G.0 is required to use this capability.

TASK CONTROL

The functions in this group control the opening, registering, and closing of tasks. The specific functions are:

- **sn90_open()**.
- **sn90_register()**.
- **sn90_close()**.

sn90_open()

The **sn90_open()** function names the issuing task and tells the task manager what that name is. Other tasks in the simulation network will use this name when referring to this task. This function must be the first simulation network API function issued by a task. Before issuing this function, insure that the task manager is running.

sn90_register()

The **sn90_register()** function assigns the issuing task to a particular simulation. All tasks used in the same simulation must register using the same simulation name. It is important to use the correct simulation name because the simulation network can support multiple simulations sharing the same ethernet network. This function must be the second simulation network API function issued by a task. Before issuing this function, insure that the simulation manager is running.

sn90_close()

The **sn90_close()** function terminates the connection of the issuing task to the simulation network. This function provides an orderly shutdown of the network connections.

SIMULATION CONTROL

The functions in this group control the execution of a simulation. The specific functions are:

- **sn90_run()**.
- **sn90_run_ack()**.
- **sn90_get_run()**.
- **sn90_freeze()**.
- **sn90_freeze_ack()**.
- **sn90_end_simulation()**.

sn90_run()

The **sn90_run()** function puts the simulation into run mode. In run mode, control modules execute their control algorithms and the simulation produces the required simulated behavior. This function also specifies the execution rate of the simulator relative to real time.

sn90_run_ack()

The **sn90_run_ack()** function should be issued by every task that receives a run command from the simulation manager. This function tells the simulation manager that the task is in run mode.

sn90_get_run()

The **sn90_get_run()** function returns the execution rate specified in the **sn90_run()** function.

sn90_freeze()

The **sn90_freeze()** function puts the simulator into freeze mode. In freeze mode, control modules stop executing control algorithms and the simulation stops producing the required behavior. The tasks still communicate notification messages.

sn90_freeze_ack()

The **sn90_freeze_ack()** function should be issued by every task that receives a freeze command from the simulation manager task. This function tells the simulation manager that the task is in freeze mode.

sn90_end_simulation()

The **sn90_end_simulation()** function sends a message to all tasks in the simulation telling them to shut down or close. USM and MFP modules respond by going to a not-initialized state waiting to start up. All other tasks should acknowledge receipt of the message, complete any required operations, and close down. When all tasks have closed down, the simulation manager task notifies the task that issued this function that all tasks in the simulation have closed. At this point, the simulation manager shuts down and the task closes down.

POINT AND MODULE INITIALIZATION

The functions in this group establish points, USM modules, USM module I/O modules, and USM points. A communication connection is specified by establishing the two end points of the connection. This is accomplished by using the **sn90_establish_USM_point()** or **sn90_establish_USM_remote_point()** function for the INFI 90 OPEN end of the connection and by using the **sn90_establish_point()** function for the application task end of the connection. The simulation network needs to be provided with the USM configuration information. The establish USM functions provide this information.

When starting up a simulation, only one task from that simulation should issue the functions that establish USM modules. The task can be a process model, simulation control application, or a task whose only purpose is to provide this information.

USM modules, I/O modules and points must be established in that order. Before a point can be established, its I/O module

must be established, and before an I/O module can be established, its USM module must be established. Establishing a process model point can take place before or after establishing USM information. The specific functions are:

- **sn90_establish_point()**.
- **sn90_establish_point_done()**.
- **sn90_establish_USM()**.
- **sn90_establish_USM_iomodule()**.
- **sn90_establish_USM_point()**.
- **sn90_establish_USM_done()**.
- **sn90_establish_USM_remote_iomodule()**.
- **sn90_establish_USM_remote_point()**.

sn90_establish_point()

A client task must issue an **sn90_establish_point()** function for every point value it needs to communicate to other tasks. The simulation network supports communications between tasks, between USM modules, and between tasks and USM modules.

The **sn90_establish_point()** function supplies the required information about a point to the simulation manager. This information includes the point name, I/O interval, I/O direction, point type, value address, point handle, and status address. All tasks must refer to the same point by the same point name.

The I/O interval provides the number of milliseconds between updates for each point. For optimum network performance, the number of different I/O intervals should be determined such that there is a balance between the need to update values at a particular frequency and the desire to keep the number of value messages sent out on the network to a reasonable level. When establishing the same point for both a transmit task and receive task, the I/O interval of the receive point determines the frequency of data exchange. The I/O interval does not solely determine when point values are exchanged between tasks. The update of point values is accomplished by providing the API services the I/O interval for each point and then by issuing the **sn90_update_values()** function frequently enough to insure the desired I/O update rate.

The value address provides the address of the memory location of the point value. A zero value address prevents direct access of the task memory by the API services. In this case, point values to be exchanged are temporarily stored in the point handle structure (using the **sn90_read_value()** and **sn90_write_value()** functions) until the **sn90_update_values()** function is issued. A non-zero value address allows direct access of memory locations. In this case, point values are automatically read from and written to the API services. Normally, the value address should not be zero.

The status address is the address of the memory location provided by the task into which the API services write the current I/O status of the point. If this field is set to zero, it is not used and memory does not need to be allocated for point status.

sn90_establish_point_done()

The **sn90_establish_point_done()** function tells the simulation manager that all I/O points have been established for that task. After issuing this function, no more **sn90_establish_point()** functions will be accepted. If this function is not issued, the I/O exchange of the points established will not occur.

sn90_establish_USM()

The **sn90_establish_USM()** function establishes a USM module and tells the simulation manager the network address of the USM module.

sn90_establish_USM_iomodule()
sn90_establish_USM_remote_iomodule()

The **sn90_establish_USM_iomodule()** and **sn90_establish_USM_remote_iomodule()** functions establish an I/O module within the USM module. One of these functions must be issued for each simulated, actual, or virtual I/O module in the simulation network. Use the I/O module presence field to designate if the I/O module is actual, simulated, or virtual. Virtual I/O modules are a means for the task to access MFP module function block output values. The use of virtual I/O modules does not require any special MFP module code configuration. There can be up to ten virtual I/O modules for an MFP module. The virtual I/O module must be assigned an unused I/O expander bus address.

sn90_establish_USM_point()
sn90_establish_USM_remote_point()

The **sn90_establish_USM_point()** and **sn90_establish_USM_remote_point()** functions establish a point in an I/O module. The point type field indicates if the point is analog or digital. The point type must match the point type expected by the associated MFP function code. Virtual I/O modules are defined as having points zero through seven as analog points and points eight through 15 as digital points. The block number field only pertains to virtual I/O modules. It provides the MFP function block number from which the task receives point values.

sn90_establish_USM_done()

The **sn90_establish_USM_done()** function tells the simulation manager that all USM initialization information has been provided. If this function is not issued, the USM modules will not receive their initialization information. After this function is issued, any further attempts to issue USM initialization information will be ignored.

MESSAGES

A task must handle messages that come from the simulation network. The functions in this group acquire and acknowledge messages. The specific functions are:

- **sn90_get_message()**.
- **sn90_ack_message()**.

sn90_get_message()

A task must follow a strict procedure when handling messages because the API services depend on this procedure so that they can manage the message buffers. A task must issue the **sn90_get_message()** function to get the message code of the next available message. The appropriate response to these message codes is detailed in Section 8. Messages that contain data other than the message code and message status require the task to issue the appropriate function to obtain the values contained in the message.

sn90_ack_message()

Use the **sn90_ack_message()** function to acknowledge notification messages that require acknowledgment.

POINT I/O VALUES

The functions in this group read, write, update, and override I/O values. The specific functions are:

- **sn90_read_value()**.
- **sn90_write_value()**.
- **sn90_update_values()**.
- **sn90_override_add()**.
- **sn90_override_clear()**.

I/O values are transmitted in the engineering units specified by the function code in the MFP module. The MFP module in simulation mode bypasses the conversion of field signals to engineering units. The process simulation task should output values according to the engineering units specified.

sn90_read_value()

The **sn90_read_value()** function returns a value from a data area accessible to the API services to the task. The API services store received values from other tasks and provide them to the task when this function is issued. This function should be used to input values if the point was established (using the **sn90_establish_point()** function) with a zero value address. A non-zero value address tells the API services to automatically transfer received values to the memory area of the task.

sn90_write_value()

The **sn90_write_value()** function passes a point value from the memory area of the task to a data area accessible to the API services. The API services store the value until a task issues the **sn90_update_values()** function and the I/O interval for that point is exceeded. This function should be used to export values if the point was established (using the **sn90_establish_point()** function) with a zero value address. A non-zero value address tells the API services to automatically transfer values from the task to the API services.

sn90_update_values()

The **sn90_update_values()** function causes the API services to process any received import and export requests. The API services queue up received import data until this function is issued. If a task fails to issue this function, export values will not be sent out on the simulation network and import values received from the simulation network will not be forwarded to the task. For the API services to export a value, the I/O interval for the point must have been exceeded ***and*** the **sn90_update_values()** function must be issued.

Point values are transmitted according to the I/O groups defined. All points in the same I/O group are sent in the same I/O values message. The sending of I/O values is not exception based. The values messages have a fixed size that is determined at start-up.

sn90_override_add()

The **sn90_override_add()** function tells the API services of a task to override the output value of a point with another value. The task that generates the original output value of the point is not aware that the value is being overridden. All destinations of the point value will receive the override value.

One use for overrides is for checking out connections in a simulation network. To do this, a technician overrides a point

value and then verifies the value is received by the destination task.

Another use for overrides is to align the control system and the simulation when being connected. To suddenly connect the control system to the simulation could numerically upset the simulation system so that many values would go out of range and the system would never return to normal operation. In this instance, overrides are used to numerically isolate the control system and simulation while an engineer manipulates both control system and simulation so that one point value at a time is approximately equal to its override value. When the override value and the actual value of a point are approximately equal, that override should be cleared. Repeat this process until all overrides are cleared.

If the value of a point controlled by an application task is overridden, the application task will not receive any notification of the override.

If the value of a point controlled by an MFP module is overridden, any application tasks receiving that point will receive the overridden value, but will not receive any notification that the value has been overridden. Overriding a value controlled by an MFP module will have different effects in the module, depending on the point type and the function code of the block controlling the point. The effects are as follows:

- Digital point: No effect.
- Analog point controlled by function code 79 block: The feedback output from the block for the affected channel will contain the overridden value. If the quality check value was set to SN90_QUALITY, the feedback output will be put into bad quality and a *Channel failure/out of range* problem for the affected block will appear in the module's problem report list.
- Analog point controlled by function code 149 block: The feedback output from the block for the affected channel will be frozen at the last non-overridden value. If the quality check value was set to SN90_QUALITY, the feedback output will be put into bad quality and a *Channel failure/out of range* problem for the affected block will appear in the module's problem report list.
- Analog point controlled by function code 150 block: The first output of the function code will be set to the override value. The output will remain in good quality regardless of the quality check value specified in the **sn90_override_add()** function.

sn90_override_clear()

The **sn90_override_clear()** function clears the override value of a point.

INFI 90 OPEN MALFUNCTION SIMULATION

The functions in this group add or clear INFI 90 OPEN hardware malfunctions. The specific functions are:

- **sn90_channel_malf_add()**.
- **sn90_channel_malf_clear()**.
- **sn90_iomodule_malf_add()**.
- **sn90_iomodule_malf_clear()**.
- **sn90_remote_iomodule_malf_add()**.
- **sn90_remote_iomodule_malf_clear()**.

Because channel malfunctions are specified by point name rather than I/O module address, separate functions to manipulate channel malfunctions in remotely mounted I/O modules are not required.

The effects of malfunctions on point data within MFP modules and application programs are as follows:

IMASO01 This is an analog output I/O module. Values are sent from the MFP control module to the client program. The MFP interface is a function code 149 block. This block features feedback outputs that report the values being sent out to the field through the I/O module.

An I/O module malfunction freezes all values seen by the client. All feedback outputs from the function code 149 block are also frozen and they are put into bad quality.

A channel malfunction freezes the associated point in the MFP control module and puts it into bad quality, but data continues to be sent to the client as though the malfunction did not exist.

IMASI02, IMASI03, and IMFBS01 These are analog input I/O modules. Values are sent from the client program to the MFP control module. A channel malfunction puts the associated output value in the MFP control module into bad quality, but the value itself continues to be updated as the client sends changes. No other points in the same MFP block are affected - they continue to operate normally.

IMDSM03 This is a digital input I/O module. Values are sent from the client program to the MFP control module. The I/O module has 16 channels and can feed data into two function code 84 blocks. A channel malfunction freezes all outputs of the function code 84 block that contains the affected channel and puts

them into bad quality. A second function code 84 connected to the same I/O module is not affected.

IMDSM04 This I/O module accepts analog values from a client program into an MFP control module. An I/O module malfunction freezes all outputs in the MFP control module and changes the I/O module status output of all associated blocks from 0 to 1. A channel malfunction freezes the output and changes the I/O module status of the affected channel only. All other channels continue to operate normally.

IMDSM05-O The IMDSM05 I/O module has two eight channel groups and either group can be configured for input or output. The IMDSM05-O I/O module represents a IMDSM05 with both groups configured to send data from an MFP control module to a client program. An I/O module malfunction freezes all values that the client program received. A channel malfunction freezes all values sent by the function code 83 block that was associated with the malfunctioning channel. Values coming from a second function code 83 associated with the same I/O module are unaffected.

IMDSM05-I This is a IMDSM05 I/O module configured to receive data from the client into the MFP control module on both groups of channels. An I/O module malfunction puts all MFP points into bad quality and holds them at current values. A channel malfunction puts all points from the same MFP block as the affected channel into bad quality and freezes their values. A second block driven by the same I/O module is unaffected.

IMDSM05-IO and IMDSM05-OI These are IMDSM05 I/O modules configured to receive data from the client on one group of channels and to send data to the client from the other group. In both cases, I/O modules freeze all data and put input points in the MFP control module into bad quality. Channel malfunctions affect all values associated with the malfunctioning channel's block and leave channels controlled by the other block unaffected.

IMDSO01, IMDSO02 and IMDSO03 These are digital I/O modules that send up to eight values from an MFP control module to a client. I/O module malfunctions and channel malfunctions both freeze all values being received by the client.

IMDSO04 This is a 16-channel digital I/O module that sends values from an MFP control module to a client. I/O module malfunctions freeze all outputs. Channel malfunctions freeze all points associated with the MFP block driving the malfunctioning channel. Data from a second block is unaffected.

IMDSI01 and IMDSI02 These are 16-channel digital I/O modules that accept values into an MFP control module from a client. An I/O module malfunction freezes all outputs and puts them into bad quality. A channel malfunction freezes all outputs from the affected function code 84 block and puts them into bad quality. A second

function code 84 driven by the same I/O module is not affected.

IMCIS02 and IMQRS02 These I/O modules handle both analog and digital values going in both directions. An I/O module malfunction puts all values coming into the MFP control module from the client into bad quality and freezes them. Feedback outputs showing data going from the MFP control module to the client also go into bad quality but their values continue to update.

A channel malfunction on an analog or digital point coming into the MFP control module from the client causes the MFP point to freeze and go into bad quality. No other channels are affected.

Channel malfunctions on data being sent from the MFP control module to the client have no effect. The data received by the client program continues to update and the feedback output for analog data remains in good quality.

IMFCS01 This I/O module receives a single value from the client program into the MFP control module. I/O module and channel malfunctions both cause the MFP point to freeze and go into bad quality.

IMHSS01 This I/O module sends a single value from the MFP control module to the client program. I/O module and channel malfunctions both cause the value received by the client to freeze.

IMSET01 and IMSED01 Sequence of events simulation is provided by offering 16 input channels.

Virtual I/O modules Attempting to establish a malfunction of either type on a virtual I/O module causes the USM module to respond with an error status 11, which is *SLAVE NOT SIMULATED*.

sn90_channel_malf_add()

The **sn90_channnel_malf_add()** function activates a channel malfunction (set point to bad quality) for the specified point.sn90_channel_malf_clear().

The **sn90_channel_malf_clear()** function deactivates the channel malfunction.

sn90_iomodule_malf_add()
sn90_remote_iomodule_malf_add()

The **sn90_iomodule_malf_add()** and **sn90_remote_iomodule_malf_add()** functions activate a malfunction at the control module or I/O module channel level. The malfunctions are referred to by malfunction codes. The codes are:

- Code 1 causes a no response or wrong type malfunction.
- Code 2 causes a calibration malfunction.
- Code 3 causes a channel failure or out of range malfunction.
- Code 4 causes a counter overflow malfunction.
- Code 5 causes a blown fuse malfunction.

An I/O module malfunction will cause all output values from blocks associated with the malfunctioning I/O module to go into bad quality if they can. Values of those points will remain at the last known good value. Output values whose purpose is to report I/O module status will show bad status. The types of malfunctions available, along with the modules and function codes that support them, are listed in Table 4-1.

Table 4-1. Possible I/O Module Malfunctions

I/O Module	Function Codes Required	Malfunction Code					Defaults Supplied By
		1	2	3	4	5	
IMASI02, IMFBS01	132	X	X	X[1]			N/A
IMASI03	215, 216, or 217	X	X	X[6]			N/A
IMASO01	149[2]	X	X	X[1,5]			MFP module
IMCIS02, IMCIS12, IMQRS02	79[2]	X	X	X[1,5]			Dipswitches
IMDSI02, IMDSI12/13/14/15	84 or 114	X[3,7]					N/A
IMDSM04	102, 103, 104, or 109	X		X[1]	X		N/A
IMDSM05	83, 84, 114, or 115	X[3,7]					Dipswitches
IMDSO01/02/03, IMDSO15	83 or 115	X[3,7]				X	MFP module
IMDSO04, IMDSO14	83 or 115	X[3,7]				X	MFP module
IMFCS01	145	X		X[1]			N/A
IMHSS01	150[2]	X[7]					Intrinsic[4]
IMRIO02	146	X[8]					N/A
IMSET01, IMSED01	242	X		X[1,9]			N/A

NOTES:

1. Channel malfunction generated in response to sn90_channel_malf_add() function call.
2. This function code reads feedback values during start-up and override.
3. Grouped channels: all 8 channels in the affected group will show bad quality after a call to sn90_channel_malf_add() for any channel in the group.
4. If this module loses its control module, it enters emergency manual mode (hold).
5. This malfunction will be reported if an analog output point from this module is overridden.
6. A call to sn90_channel_malf_add() for one of this module's channels will result in this malfunction being reported for the affected FC 216 block. A call to sn90_iomodule_malf_add() for this module will result in this malfunction being reported for the modules FC 215 block.
7. This malfunction code will be reported if sn90_channel_malf_add() is called for any channels in this I/O module type.
8. A malfunction can only be simulated for an IMRIO02 configured as a Remote Master Processor. All channels of all I/O modules communicating through the simulated RMP module will report a no response/wrong type error when the RMP is malfunctioning.
9. Grouped channels: all 16 channels in this I/O module will show zero and bad quality after a call to sn90_channel_malf_add() for any channel in the I/O module.

sn90_iomodule_malf_clear()
sn90_remote_iomodule_malf_clear()

The **sn90_iomodule_malf_clear()** and **sn90_remote_iomodule_malf_clear()** functions clear a malfunction that was activated using the **sn90_iomodule_malf_add()** or **sn90_remote_iomodule_malf_add()** functions.

SNAPSHOT DATA

A snapshot is a set of data that defines the state of the simulation at a certain time. A snapshot is saved so that the simulation can be restored at a later time to the saved state.

The functions in this group save and restore snapshot data. The specific functions are:

- **sn90_snapshot_save().**
- **sn90_snapshot_save_ack().**
- **sn90_snapshot_restore().**
- **sn90_snapshot_restore_dynamic().**
- **sn90_snapshot_restore_task().**
- **sn90_snapshot_restore_task_dyn().**
- **sn90_snapshot_restore_ack().**
- **sn90_get_snapshot_save().**
- **sn90_get_snapshot_restore().**
- **sn90_snapshot_delete().**
- **sn90_snapshot_delete_ack().**
- **sn90_get_snapshot_delete().**

The snapshot save of the MFP module state is managed by the simulation manager. Exchange of point values continues between an MFP module and the process model task during a snapshot save collection if the simulator is in run mode.

During simulator start-up, the simulation manager automatically collects the configuration data (function block configuration) from each MFP module and stores it in a directory specified during the simulation manager start-up. During a requested snapshot save, the simulation manager collects only the dynamic data from the MFP modules and does not recollect the configuration data. A copy of the MFP module configuration data is copied into the save directory as part of a snapshot save request. Tuning specification changes to the MFP module function codes are updated automatically in the configuration snapshot file.

A snapshot directory must be created and the name of the directory must be used in the snapshot save request. The simulation manager forwards the snapshot request to all USM modules and application tasks connected to the simulation. The simulation manager collects the data from the USM modules and writes the information to the appropriate files in the

specified snapshot directory. A directory can contain only one set of snapshot data. If a set of snapshot data already exists in the snapshot directory, the simulation manager will delete the existing data and save the new snapshot data. The snapshot save operation saves the following types of data:

- Dynamic data (such as block outputs) from the MFP modules.
- Point and I/O module configuration data and related information from the USM modules.
- C, batch, and data files from the MFP modules.
- Active simulated INFI 90 OPEN malfunctions.
- Active overrides.

The time required to complete a snapshot save or restore is system dependent and can be as great as three minutes for a system containing MFP modules with large configurations. The MFP module continues to execute its control logic and exchange I/O values during snapshot saves if the system is in run mode.

The application task does not directly write or read the snapshot files. The simulation manager sends a snapshot save message to application tasks. Application tasks are required to send an acknowledge message back to the simulation manager when any snapshot operations they need to perform are complete.

The following types of files are created in the snapshot directory:

content.snp - contains a map of the contents of the snapshot data in this directory.

taskid.st - contains static data from the MFP module, including control logic configuration and tunable specifications. One file of this type is created for every USM module connected in the simulator. ***taskid*** is the name provided in the **sn90_establish_USM()** function.

taskid.f# - contains a C, batch, or data file from a MFP module. One file of this type is created for every C file in every MFP module that is connected in the simulator. ***taskid*** is the name provided in the **sn90_establish_USM()** function. **#** is a numeric value assigned to the file (e.g., ***taskid.f0***, ***taskid.f1***, etc.).

taskid.dy - contains dynamic data such as block outputs. One file of this type is created for every USM module and application task connected in the simulator. ***taskid*** is the name

provided in the **sn90_establish_USM()** function (for USM modules) or the **sn90_open()** function (for application tasks).

A checksum based on the MFP function code number and function code block number is stored in the ***.st***, ***.dy***, and ***content.snp*** files during snapshot save operations. During snapshot restore operations, the checksum from each of the files is cross checked and compared to the checksum of the control logic currently in the MFP module. If any of these checksums do not match, the snapshot restore operation for that MFP module fails and an error is logged. The simulation manager will not restore a snapshot that would modify the current control logic configuration (except tunable specifications) in that MFP module.

sn90_snapshot_save()

The **sn90_snapshot_save()** function directs the simulation manager to save a snapshot of all USM/MFP module pairs and to issue snapshot commands to all tasks in the simulation.

sn90_get_snapshot_save()

The **sn90_get_snapshot_save()** function returns to the task the snapshot ID that was specified in the **sn90_snapshot_ save()** function.

sn90_snapshot_save_ack()

The **sn90_snapshot_save_ack()** notifies the simulation manager that an application task has completed its snapshot save processing. This function must be called after a snapshot save command message is received even if the application task does not do anything in response to that message.

sn90_snapshot_restore()

The **sn90_snapshot_restore()** function directs the simulation manager to restore USM and MFP modules to the saved condition and to issue a restore command to all tasks in the simulation.

sn90_snapshot_restore_dynamic()

The **sn90_snapshot_restore_dynamic()** function restores the same information as the **sn90_snapshot_restore()** function, except the MFP module static configuration data is not restored. The function block tuning specifications are stored in the static configuration file. The effect of this operation is to restore block output data to a known configuration, but to leave block specifications unchanged.

sn90_snapshot_restore_task()

The **sn90_snapshot_restore_task()** function directs the simulation manager to restore one task. A task can be an application task or a USM/MFP module pair.

sn90_get_snapshot_restore()

The **sn90_get_snapshot_restore()** function returns to the task the snapshot ID that was specified in the **sn90_snapshot_restore()** function.

sn90_snapshot_restore_task_dyn()

The **sn90_snapshot_restore_task_dyn()** function restores the same information as the **sn90_snapshot_restore_task()** function, except the MFP module static configuration data is not restored.

sn90_snapshot_restore_ack()

The **sn90_snapshot_restore_ack()** notifies the simulation manager that an application task has completed its snapshot restore processing. This function must be called after a snapshot restore command message is received, even if the application task does not do anything in response to that message.

sn90_snapshot_delete()

The **sn90_snapshot_delete()** function directs the simulation manager to delete a snapshot directory and its contents. The simulation manager will also command all client programs to delete the same snapshot.

sn90_snapshot_delete_ack()

The **sn90_snapshot_delete_ack()** function notifies the simulation manager that an application task has completed its snapshot deletion processing. This function must be called after a snapshot delete command is received, even if the application task does not do anything in response to that message.

sn90_get_snapshot_delete()

The **sn90_get_snapshot_delete()** function returns to the task the snapshot identifier that was specified in the **sn90_snapshot_delete()** function.

SIMULATION NETWORK QUERYING

The functions in this group allow a task to query the simulation manager for information about a running simulation. This information consists of task IDs and statuses, point names, active module malfunctions, active overrides, and snapshot file information.

Collect Functions

When a task issues a collect function, a list of information is compiled. When the list is complete, the task receives a notification message. The queries are implemented in such a manner that the requesting task does not wait for the response. This method prevents tasks that act as both a user interface and process simulation from spending large amounts of time waiting for data to be collected and returned over the simulation network. The time required to collect information cannot be guaranteed to be less than the minimum supported calculation interval (250 milliseconds). The collect functions consist of:

- **sn90_collect_channel_malf()**.
- **sn90_collect_iomodule_malf()**.
- **sn90_collect_overrides()**.
- **sn90_collect_pointnames()**.
- **sn90_collect_simulation_ID()**.
- **sn90_collect_snapshot()**.
- **sn90_collect_task_status()**.
- **sn90_collect_remote_channel_malf()**.
- **sn90_collect_remote_iomodule_malf()**.

The remote functions are provided to offer differentiation between local and remote malfunctions and to ensure that it is possible to build a complete simulation system using only the remote function calls. The collect remote malfunction functions allow the user to specify whether to collect only malfunctions in local I/O modules, only malfunctions in remote I/O modules, or all malfunctions.

Next Functions

The API services queue the information list and provide one record at a time when the appropriate next function is issued. Issue the next function until the SN90_EOF status is returned. The next functions consist of:

- **sn90_next_channel_malf()**.
- **sn90_next_iomodule_malf()**.
- **sn90_next_override()**.
- **sn90_next_pointname()**.
- **sn90_next_simulation_ID()**.
- **sn90_next_snapshot_task()**.

- **sn90_next_task_status()**.
- **sn90_next_remote_iomodule_malf()**.

The **sn90_next_remote_iomodule_malfunction()** function provides I/O module address data in a SlaveAddress structure. This is optional for local I/O modules and is required for remote I/O modules. Since there is no difference in data provided for channel malfunctions for local and remote I/O modules, an **sn90_next_remote_channel_malfunction()** function is not provided.

SIMULATION TIME MANAGEMENT

The functions in this group acquire, set, and acknowledge the simulation time. The specific functions are:

- **sn90_set_simulation_time()**.
- **sn90_simulation_time_ack()**.
- **sn90_get_simulation_time()**.

sn90_set_simulation_time()

The **sn90_set_simulation_time()** function sends the simulation time notification message to all other tasks in the simulation.

sn90_simulation_time_ack()

Tasks must issue the **sn90_simulation_time_ack()** function to acknowledge receipt of the simulation time notification message.

sn90_get_simulation_time()

When an SN90_SIMULATION_TIME notification message is received, each task should issue the **sn90_get_simulation_time()** function to receive the simulation time parameters. After receiving the parameters, tasks should set their simulation time to match the parameters.

LOG FILE

The simulation network stores all pertinent command requests in a common log file with a temporary file name. The log file contains simulation network information that is stored there by the simulation manager. The **sn90_close_transaction_log()** function instructs the simulation network to close and rename the log file. A new temporary log file is opened at this time. Log files are in ASCII format. Log files are available for viewing, printing, and are accessible to other tasks. Refer to ***LOG FILE*** in Section 6 for information on interpreting the log file and a print out of a sample log file.

SECTION 5 - FUNCTION LIBRARY

INTRODUCTION

This section describes and lists the application program interface (API) functions available to application tasks.

HEADER FILE

The ***sn90_api.h*** header file must be included in the application program. This header file is the only one needed and is supplied with the simulation network API software.

FORMAT

All API functions must be called or issued by the application task using the following format:

status = *sn90_command_name()*

FUNCTIONS

The API functions can be categorized into several groups:

- Task control.
- Simulation control.
- Point and module initialization.
- Messages.
- Point I/O values.
- INFI 90 OPEN malfunction simulation.
- Snapshot data.
- Simulation network querying.
- Simulation time management.
- Log file.

Tables 5-1 and 5-2 provide quick reference listings of the API functions. Detailed information about each function follows.

Table 5-1. API Functions Listed by Type

Type	Function	Purpose
Task control	sn90_open() sn90_register() sn90_close()	Open, register, and close tasks.
Simulation control	sn90_end_simulation() sn90_freeze() sn90_freeze_ack() sn90_get_run() sn90_run() sn90_run_ack()	Control the execution of simulations.

Table 5-1. API Functions Listed by Type (Continued)

Type	Function	Purpose
Point and module initialization	sn90_establish_point() sn90_establish_point_done() sn90_establish_USM() sn90_establish_USM_done() sn90_establish_USM_iomodule() sn90_establish_USM_remote_iomodule() sn90_establish_USM_point() sn90_establish_USM_remote_point()	Establish I/O points, USM points, and USM modules.
Messages	sn90_ack_message() sn90_get_message()	Acquire and acknowledge messages.
Point I/O values	sn90_override_add() sn90_override_clear() sn90_read_value() sn90_update_values() sn90_write value()	Read, write, override, and update values.
INFI 90 OPEN malfunction simulation	sn90_channel_malf_add() sn90_channel_malf_clear() sn90_iomodule_malf_add() sn90_remote_iomodule_malf_add() sn90_iomodule_malf_clear() sn90_remote_iomodule_malf_clear()	Simulate INFI 90 OPEN malfunctions.
Snapshot data	sn90_get_snapshot_delete() sn90_get_snapshot_restore() sn90_get_snapshot_save() sn90_snapshot_delete() sn90_snapshot_delete_ack() sn90_snapshot_restore() sn90_snapshot_restore_ack() sn90_snapshot_restore_dynamic() sn90_snapshot_restore_task() sn90_snapshot_restore_task_dyn() sn90_snapshot_save() sn90_snapshot_save_ack()	Save and restore snapshot data.
Simulation network querying	sn90_collect_channel_malf() sn90_collect_remote_channel_malf() sn90_collect_iomodule_malf() sn90_collect_remote_iomodule_malf() sn90_collect_overrides() sn90_collect_pointnames() sn90_collect_simulation_ID() sn90_collect_snapshot() sn90_collect_task_status() sn90_next_channel_malf() sn90_next_iomodule_malf() sn90_next_remote_iomodule_malf() sn90_next_override() sn90_next_pointname() sn90_next_simulation_ID() sn90_next_snapshot_task() sn90_next_task_status()	Acquire network information.
Simulation time management	sn90_get_simulation_time() sn90_set_simulation_time() sn90_simulation_time_ack()	Acquire and set the simulation time.

Table 5-1. API Functions Listed by Type (Continued)

Type	Function	Purpose
Log file	sn90_close_transaction_log()	Close the transaction log.

Table 5-2. API Functions Listed Alphabetically

Function	Type
sn90_ack_message()	Messages
sn90_channel_malf_add()	INFI 90 OPEN malfunction simulation
sn90_channel_malf_clear()	
sn90_close()	Task control
sn90_close_transaction_log()	Log file
sn90_collect_channel_malf()	Simulation network querying
sn90_collect_iomodule_malf()	
sn90_collect_overrides()	
sn90_collect_pointnames()	
sn90_collect_remote_channel_malf()	
sn90_collect_remote_iomodule_malf()	
sn90_collect_simulation_ID()	
sn90_collect_snapshot()	
sn90_collect_task_status()	
sn90_end_simulation()	Simulation control
sn90_establish_point()	Point and module initialization
sn90_establish_point_done()	
sn90_establish_USM()	
sn90_establish_USM_done()	
sn90_establish_USM_iomodule()	
sn90_establish_USM_point()	
sn90_establish_USM_remote_iomodule()	
sn90_establish_USM_remote_point()	
sn90_freeze()	Simulation control
sn90_freeze_ack()	
sn90_get_message()	Messages
sn90_get_run()	Simulation control
sn90_get_simulation_time()	Simulation time management
sn90_get_snapshot_delete()	Snapshot data
sn90_get_snapshot_restore()	
sn90_get_snapshot_save()	
sn90_iomodule_malf_add()	INFI 90 OPEN malfunction simulation
sn90_iomodule_malf_clear()	

Table 5-2. API Functions Listed Alphabetically (Continued)

Function	Type
sn90_next_channel_malf()	Simulation network querying
sn90_next_iomodule_malf()	
sn90_next_override()	
sn90_next_pointname()	
sn90_next_remote_iomodule_malf()	
sn90_next_simulation_ID()	
sn90_next_snapshot_task()	
sn90_next_task_status()	
sn90_open()	Task control
sn90_override_add()	Point I/O values
sn90_override_clear()	
sn90_read_value()	
sn90_register()	Task control
sn90_remote_iomodule_malf_add()	INFI 90 OPEN malfunction simulation
sn90_remote_iomodule_malf_clear()	
sn90_run()	Simulation control
sn90_run_ack()	
sn90_set_simulation_time()	Simulation time management
sn90_simulation_time_ack()	
sn90_snapshot_delete()	Snapshot data
sn90_snapshot_delete_ack()	
sn90_snapshot_restore()	
sn90_snapshot_restore_ack()	
sn90_snapshot_restore_dynamic()	
sn90_snapshot_restore_task()	
sn90_snapshot_restore_task_dyn()	
sn90_snapshot_save()	
sn90_snapshot_save_ack()	
sn90_update_values()	Point I/O values
sn90_write_value()	

sn90_ack_message()

messages

PURPOSE: Acknowledges the receipt of the following messages and tells the simulation manager that the application task has successfully completed any required operations:

SN90_RUN
SN90_FREEZE
SN90_SNAPSHOT_SAVE
SN90_SNAPSHOT_RESTORE
SN90_SIMULATION_TIME
SN90_END_OF_SIMULATION

FORMAT: **short sn90_ack_message()**

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: Proper execution of this function returns SN90_NORMAL.

sn90_channel_malf_add()

INFI 90 OPEN malfunction simulation

PURPOSE: Activates the INFI 90 OPEN I/O module channel malfunction.

FORMAT: **short sn90_channel_malf_add(***task_ID***, *point_name*)**

Parameter	Type	Description
task_ID	char*	Name of the USM task to which the malfunction is being added. Must be a null terminated string. Maximum length of 16 alphanumeric characters. An empty string (“”) can be used for the task_ID. In this case, all tasks are searched and the first point found with the specified name is placed into malfunction.
point_name	char*	Name of the point for which the malfunction is to occur. Must be a null terminated string. Maximum length of 32 alphanumeric characters.

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR
SN90_NOT_REGISTERED

REMARKS: A successful execution of this function results in an SN90_MALFUNCTION_ACK notification message indicating that the simulation manager has received the request.

sn90_channel_malf_clear()

INFI 90 OPEN malfunction simulation

PURPOSE:

Deactivates the channel malfunction activated by the **sn90_channel_malf_add()** function.

FORMAT:

short sn90_channel_malf_clear(*task_ID***,** *point_name***)**

Parameter	Type	Description
task_ID	char*	Name of the task from which the malfunction is being removed. Must be a null terminated string. Maximum length of 16 alphanumeric characters. An empty string (“”) can be used for the task_ID. In this case, all tasks are searched and the first point found with the specified name has an existing malfunction removed.
point_name	char*	Name of the point that will be cleared of the malfunction. Must be a null terminated string. Maximum length of 32 alphanumeric characters.

RETURNS:

SN90_NORMAL
SN90_SEND_MSG_ERROR
SN90_NOT_REGISTERED

REMARKS:

A successful execution of this function results in an SN90_MALFUNCTION_ACK notification message indicating the simulation manager has received the request.

sn90_close()

task control

PURPOSE: Disconnects a task from the simulation network.

FORMAT: **void sn90_close(***delay_interval***)**

Parameter	Type	Description
delay_interval	float	Time interval in seconds between issuing this function and the disconnection of the task. A value of 1.0 is normally acceptable.

RETURNS: None.

REMARKS: This function provides an orderly shutdown of the network connections for that task.

sn90_close_transaction_log()

log file

PURPOSE:

Closes the transaction log file and renames it to the specified file name.

FORMAT:

short sn90_close_transaction_log(*directory_name***,** *file_name***)**

Parameter	Type	Description
directory_name	char*	Name of the directory that will contain the log file. Must be a null terminated string. Maximum length of 64 alphanumeric characters.
file_name	char*	Name of the stored log file. Must be a null terminated string. Maximum length of 16 alphanumeric characters.

RETURNS:

SN90_NORMAL
SN90_SEND_MSG_ERROR
SN90_NOT_REGISTERED

REMARKS:

The simulation network stores all pertinent command requests in a common log file with a temporary file name. This function closes and renames the log file. A new log file is opened at the same time. The log files are available for viewing, printing, and are accessible to tasks. Transaction log files are in ASCII format.

sn90_collect_channel_malf()

simulation network querying

PURPOSE: Compiles a listing of all the active channel malfunctions in the simulation.

FORMAT: **short sn90_collect_channel_malf()**

RETURNS:
SN90_NORMAL
SN90_NOT_REGISTERED
SN90_SEND_MSG_ERROR

REMARKS: SN90_CHANNEL_MALFUNCTIONS_COLLECTED is provided by the **sn90_get_message()** function to the task issuing this function when all the active channel malfunctions are collected. Issue the **sn90_next_channel_malf()** function to receive the name and type of channel malfunctions collected.

sn90_collect_iomodule_malf()

simulation network querying

PURPOSE: Compiles a list of all the active I/O module malfunctions in the simulation.

FORMAT: **short sn90_collect_iomodule_malf()**

RETURNS: SN90_NORMAL
SN90_NOT_REGISTERED
SN90_SEND_MSG_ERROR

REMARKS: SN90_IOMODULE_MALFUNCTIONS_COLLECTED is provided by the **sn90_get_message()** function to the task issuing this function when all the active I/O module malfunctions are collected. Issue the **sn90_next_iomodule_malf()** function to receive the task ID, I/O module address, and type of I/O module malfunctions collected.

sn90_collect_overrides()

simulation network querying

PURPOSE: Compiles a list of all the active overrides in the simulation.

FORMAT: **short sn90_collect_overrides()**

RETURNS: SN90_NORMAL
SN90_NOT_REGISTERED
SN90_SEND_MSG_ERROR

REMARKS: SN90_TASK_OVERRIDES_COLLECTED is provided by the **sn90_get_message()** function to the task issuing this function when all the active overrides are collected. Issue the **sn90_next_override()** function to receive the name, override value, actual value, and quality of the overrides collected.

sn90_collect_pointnames()

simulation network querying

PURPOSE: Compiles a list of the names of all the points established for a task.

FORMAT: **short sn90_collect_pointnames(***task_ID***)**

Parameter	Type	Description
task_ID	char*	Name of the task for which the point names are collected. Must be a null terminated string. Maximum length of 16 alphanumeric characters.

RETURNS:
SN90_NORMAL
SN90_NOT_REGISTERED
SN90_SEND_MSG_ERROR

REMARKS: SN90_POINT_NAMES_COLLECTED is provided by the **sn90_get_message()** function to the task issuing this function when the name of all the points established for the task are collected. Issue the **sn90_next_pointname()** function to receive the task ID, name, type, I/O direction, and I/O interval of the point names collected.

sn90_collect_remote_channel_malf()

simulation network querying

PURPOSE: Compiles a listing of all active remote channel malfunctions, local channel malfunctions, or all channel malfunctions in the simulation.

FORMAT: **short sn90_collect_remote_channel_malf(***slaveClass***)**

Parameter	Type	Description
slaveClass	short	Indicates whether remote, local or all malfunctions are to be collected. Possible values include: SN90_REMOTE SN90_LOCAL SN90_ALL

RETURNS:
SN90_NORMAL
SN90_NOT_REGISTERED
SN90_SEND_MSG_ERROR

REMARKS: SN90_CHANNEL_MALFUNCTIONS_COLLECTED is provided by the **sn90_get_message()** function to the task issuing this function when all the active channel malfunctions are collected. Issue the **sn90_next_channel_malf()** function to receive the name and type of channel malfunctions collected.

> **NOTE:** Because I/O module address information is not included in channel malfunction data, a separate **sn90_next_remote_channel_malf()** function is not needed.

sn90_collect_remote_iomodule_malf()

simulation network querying

PURPOSE:

Compiles a listing of all active remote I/O module malfunctions, local I/O module malfunctions, or all I/O module malfunctions in the simulation.

FORMAT:

short sn90_collect_remote_iomodule_malf(*slaveClass***)**

Parameter	Type	Description
slaveClass	short	Indicates whether remote, local, or all malfunctions are to be collected. Possible values include: SN90_REMOTE SN90_LOCAL SN90_ALL

RETURNS:

SN90_NORMAL
SN90_NOT_REGISTERED
SN90_SEND_MSG_ERROR

REMARKS:

SN90_IOMOD_MALFUNCTIONS_COLLECTED is provided by the **sn90_get_message()** function to the task issuing this function when all the active I/O module malfunctions are collected. Issue the **sn90_next_remote_iomodule_malf()** function to receive the task ID, I/O module address (in a SlaveAddress structure), and type of I/O module malfunctions collected.

sn90_collect_simulation_ID()

simulation network querying

PURPOSE: Compiles a list of all simulations running at the time this function is issued.

FORMAT: **short sn90_collect_simulation_ID()**

RETURNS:
SN90_ERROR
SN90_NORMAL
SN90_NOT_REGISTERED

REMARKS: Use this function to determine the name of a simulation and if it is active. This function is used by tasks that are not necessarily dedicated to one particular simulation (e.g. simulation control console). After this function returns successfully, the list of simulation IDs is available. Issue the **sn90_next_simulation_ID()** function to receive the names of the active simulations. This function does not result in a notification message as do the other ***collect*** functions. Issue the **sn90_next_simulation_ID()** function to receive the collected simulation IDs.

sn90_collect_snapshot()

simulation network querying

PURPOSE: Retrieves information from the snapshot file.

FORMAT: **short sn90_collect_snapshot(***snapshot_directory***,** *snapshot_ID***)**

Parameter	Type	Description
snapshot_ directory	char*	Name of the directory containing the snapshot data. Must be a null terminated string. Maximum length of 64 alphanumeric characters.
snapshot_ID	char*	Not used. Retained for backward compatibility.

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: SN90_SNAPSHOT_COLLECTED is provided by the **sn90_get_ message()** function to the task issuing this function when the snapshot data is collected and saved. Issue the **sn90_next_ snapshot_task()** function repeatedly to receive the names and snapshot save statuses of each task included in the snapshot. Issue the **sn90_get_snapshot_time()** function to get time information associated with the snapshot data.

sn90_collect_task_status()

simulation network querying

PURPOSE: Compiles a list of the status of each task within the simulation.

FORMAT: **short sn90_collect_task_status()**

RETURNS:
SN90_NORMAL
SN90_NOT_REGISTERED
SN90_SEND_MSG_ERROR

REMARKS: SN90_TASK_STATUS_COLLECTED is provided by the **sn90_get_message()** function to the task issuing this function when the status of each task is collected. Issue the **sn90_next_task_status()** function to receive the task ID and individual statuses of the collected task statuses.

sn90_end_simulation()

simulation control

PURPOSE: Ends the simulation issuing this function.

FORMAT: **short sn90_end_simulation()**

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: SN90_END_OF_SIMULATION_DONE is provided by the **sn90_get_message()** function to the task issuing this function. The receipt of this message indicates that the simulation manager has successfully executed this function. During the end simulation operation, MFP modules are toggled to the CONFIGURE mode and then back to the EXECUTE mode. This process can take 30 to 45 seconds.

sn90_establish_point()

point and module initialization

PURPOSE: Defines a point through which an application task will exchange data over the simulation network.

FORMAT: **short sn90_establish_point(***point_name***,** *IO_interval***,** *IO_direction***,** *point_type***,** *value_address***,** *point_handle***,** *status_address***)**

Parameter	Type	Description
point_name	char*	Name used to reference a specific point. Must be a null terminated string. Maximum length of 32 alphanumeric characters.
IO_interval	long	Number of milliseconds between updates for each point. This parameter is not used if the I/O direction is set to SN90_SEND. The minimum supported interval is 250 msecs.
IO_direction	short	Direction of I/O for the point value relative to the issuing task. Available options are: SN90_RECV (0) SN90_SEND (1)
point_type	short	Type of point. Available options are: SN90_BYTE (1) (1 byte) SN90_SHORT (2) (2 bytes) SN90_LONG (3) (4 bytes) SN90_FLOAT (4) (4 bytes) SN90_DOUBLE (5) (8 bytes) The point type must match the data type used for the point in the application.
value_address	void*	Memory address of the value of a specific point. A value of 0 is used if the **sn90_read_value()** and **sn90_write_value()** functions are to be used.
point_handle	SN90_POINT*	Memory address of a data structure used to store information about a point. The application allocates this structure. A value of 0 can be used if the **sn90_read_value()** and **sn90_write_value()** functions are not used.
status_address	short*	Memory address of the I/O status of a point. The I/O status of the reception point is stored here by the API services.

RETURNS: SN90_ERROR
SN90_NORMAL

sn90_establish_point_done()

point and module initialization

PURPOSE: Notifies the simulation network that all points needed for the application task have been established.

FORMAT: **short sn90_establish_point_done()**

RETURNS: SN90_ERROR
SN90_NORMAL

REMARKS: Use this function after all points for a task have been established. I/O exchange of values for established points will not occur unless this function is used.

sn90_establish_USM()

point and module initialization

PURPOSE: Provides the simulation network with the name and network address of a USM module.

FORMAT: **short sn90_establish_USM(***internet_address***,** *task_ID***)**

Parameter	Type	Description
internet_address	char*	Network address of the USM module. Address format is *x.x.x.x* where *x* is 1 through 254. Must be a null terminated string.
task_ID	char*	Name used to reference the USM module. Must be a null terminated string.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: The network address must be unique within an ethernet network and match the address set by dipswitches on the USM module. The network address must be a valid TCP/IP address.

sn90_establish_USM_done()

point and module initialization

PURPOSE: Notifies the simulation network that all USM module information has been entered.

FORMAT: **short sn90_establish_USM_done()**

RETURNS:
SN90_NORMAL
SN90_RECEIVE_MSG_ERROR
SN90_SEND_MSG_ERROR
SN90_TIMEOUT

REMARKS: Use this function after all USM module establish functions for a simulation have been made. The USM modules will not receive initialization information if this function is not used.

The simulation manager connects to each USM module and collects the static configuration information from the MFP module. The collection of the configuration information should be allowed to complete before issuing an **sn90_run()** function. Depending on which MFP module is used, it can take 30 seconds to collect the configuration information. When the MFP module configuration has been collected for all established USM modules, an SN90_SAVE_DONE notification message is sent to each client task.

sn90_establish_USM_iomodule()

point and module initialization

PURPOSE: Defines a locally mounted I/O module connected to the USM module whose name is given by the function's first argument.

FORMAT: **short sn90_establish_USM_iomodule(***task_ID***,** *Xbus_address***,** *iomodule_presence***,** *iomodule_type***,** *virtual_update_time***)**

Parameter	Type	Description
task_ID	char*	Name of the USM module to which the I/O module is connected. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
Xbus_address	short	I/O expander bus address of the I/O module. Available addresses are 0 through 63.
iomodule_presence	short	I/O module existence. Available options are: SN90_SIMULATE (1) SN90_ACTUAL (2) SN90_VIRTUAL (3)
iomodule_type	short	I/O module type. Available options are: SN90_ASO01 (1) SN90_ASI02 (2) SN90_ASI03 (3) SN90_DSM03 (4) SN90_DSM05_O (5) SN90_DSM04 (6) SN90_DSO01 (7) SN90_DSO02 (8) SN90_DSO03 (9) SN90_DSO04 (10) SN90_DSI01 (11) SN90_DSI02 (12) SN90_CIS02 (13) SN90_QRS02 (14) SN90_FCS01 (15) SN90_HSS01 (16) SN90_DSM05_I (17) SN90_VIRT_IO (18) SN90_DSM05_IO (19) SN90_DSM05_OI (20) SN90_FBS01 (21) SN90_S0E (22) SN90_RIO02 (23)
virtual_update_time	short	For virtual I/O modules only. Number of milliseconds between MFP module updates of function block values to the USM module.

sn90_establish_USM_iomodule() *(continued)*

point and module initialization

RETURNS:

SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS:

This function must be issued for each locally mounted simulated, actual, or virtual I/O module used by the simulation.

The IMDSM05 module can be one of four types:

- DSM05_O (16 output channels).
- DSM05_I (16 input channels).
- DSM05_IO (eight input channels (zero through seven) and eight output channels (eight through 15).
- DSM05_OI (eight output channels (zero through seven) and eight input channels (eight through 15).

The IMDSM04 is an I/O module that monitors a digital signal (e.g., a frequency or pulse) but creates an analog value that describes the signal. The modeling program must generate the analog value rather than the digital signal.

If remotely mounted I/O modules are being simulated, the IMRIO02 module that will act as the Remote Master Processor must be established in the appropriate USM module using this function. Since no field data comes directly into an IMRIO02 module, no points can be established for an IMRIO02 module. Simulated IMRIO02 modules used as Remote Slave Processors do not need to be established in the simulation.

Several I/O modules have been introduced that are functionally equivalent to previously available I/O modules. Refer to Table 5-3 for types for the new modules:

Table 5-3. Types for New Modules

New I/O Module	Type
IMCIS12	SN90_CIS02 (13)
IMDSI12	SN90_DSI02 (12)
IMDSI13	SN90_DSI02 (12)
IMDSI14	SN90_DSI02 (12)
IMDSI15	SN90_DSI02 (12)
IMDSO14	SN90_DSO04 (10)
IMDSO15	SN90_DSO01 (7)

sn90_establish_USM_point()

point and module initialization

PURPOSE: Defines a point (from a locally mounted I/O module) whose value is exchanged over the simulation network.

FORMAT: **short sn90_establish_USM_point(***task_ID*, *Xbus_address*, *iomodule_channel_number*, *point_name*, *iomodule_default_status*, *point_type_ethernet*, *IO_direction*, *IO_interval*, *block_number***)**

Parameter	Type	Description
task_ID	char*	Name of the USM module that contains this point.Must be a null terminated string. Maximum length of 16 alphanumeric characters.
Xbus_address	short	I/O expander bus address of the I/O module that contains this point. Available addresses are 0 through 63.
iomodule_channel_number	short	Channel address corresponding to the function code for the particular I/O module type. Available addresses are dependent on the I/O module type. Refer to Table 5-3.
point_name	char*	Name of the I/O point. Must be a null terminated string. Maximum length of 32 alphanumeric characters.
iomodule_default_status	short	Output value of the USM module if the MFP module becomes unavailable. For I/O modules that do not get their default status from dipswitch settings, specify SN90_NA. Available options are: SN90_LOW (0) SN90_HIGH (1) SN90_HOLD (2) SN90_NA (3)
point_type_ethernet	short	Type of point. Available options are: SN90_DIGITAL (1) SN90_ANALOG (4)
IO_direction	short	Direction of data transmission relative to the MFP module. Available options are: SN90_RECV (0) SN90_SEND (1)
IO_interval	long	Time in milliseconds in which the point values should be exchanged between tasks. Minimum supported I/O interval is 250 msecs.
block_number	short	For virtual I/O module points only. MFP block number where tasks retrieve values.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: All points must be established for an I/O module even if it is not connected to a process model. For example, I/O modules having eight channels must establish all eight points on the USM module. The unused channels are given dummy point names.

Default values are used only if communications with the MFP module are interrupted. The establishment of a malfunction on an I/O module or a channel will cause the output of that channel to remain locked at its last known value.

The IMDSM04 is an I/O module that monitors a digital signal (e.g., a frequency or pulse) but creates an analog value that describes the signal. The modeling program must generate the analog value rather than the digital signal.

The iomodule_default_status needs to be specified for output channels for the following iomodule_type:

- SN90_CIS02.
- SN90_DSM05_IO.
- SN90_DSM05_O.
- SN90_DSM05_OI.
- SN90_HSS01.
- SN90_QRS02.
- SN90_VIRT_IO.

All other channels should use SN90_NA as the default status.

NOTES:

1. Default values for CIS02 and QRS02 I/O modules only apply to their analog output points.

2. Digital output defaults are low state (de-energized).

sn90_establish_USM_point() *(continued)*

point and module initialization

Table 5-4. I/O Module Channel Addresses

I/O Module Type	Channel Type	Available Addresses
SN90_ASI02	Analog input	0 - 14
SN90_ASI03	Analog input	0 - 15
SN90_ASO01	Analog output	0 - 13
SN90_CIS02	Analog input Analog output Digital input Digital output	0 - 3 4 - 5 6 - 8 9 - 12
SN90_DSI01	Digital input	0 - 15
SN90_DSI02	Digital input	0 - 15
SN90_DSM03	Digital input	0 - 15
SN90_DSM04	Analog input	0 - 7
SN90_DSM05_I	Digital input	0 - 15
SN90_DSM05_IO	Digital input Digital output	0 - 7 8 - 15
SN90_DSM05_O	Digital output	0 - 15
SN90_DSM05_OI	Digital output Digital input	0 - 7 8 - 15
SN90_DSO01	Digital output	0 - 7
SN90_DSO02	Digital output	0 - 7
SN90_DSO03	Digital output	0 - 7
SN90_DSO04	Digital output	0 - 15
SN90_FBS01	Analog input	0-14
SN90_FCS01	Analog input	0
SN90_HSS01	Analog output	0
SN90_QRS02	Analog input Analog output Digital input Digital output	0 - 3 4 - 5 6 - 8 9 - 12
SN90_SOE	Digital input	0-15
SN90_RIO02	Remote I/O	None
SN90_VIRT_IO	Analog output Digital output	0 - 7 8 - 15

sn90_establish_USM_remote_iomodule()

point and module initialization

PURPOSE:

Defines a remotely mounted I/O module connected to the USM module whose name is given by the function's first argument.

FORMAT:

short sn90_establish_USM_remote_iomodule(*task_ID***,** *address***,** *iomodule_presence***,** *iomodule_type***,** *virtual_update_time***)**

Parameter	Type	Description
task_ID	char*	Name of the USM module to which the I/O module is connected. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
address	struct Slave Address	Address of the I/O module. Refer to ***REMOTE I/O MODULE SIMULATION*** in Section 4 for a description of the SlaveAddress structure.
iomodule_presence	short	I/O module existence. Available options are: SN90_SIMULATE (1) SN90_ACTUAL (2) SN90_VIRTUAL (3)
iomodule_type	short	I/O module type. Available options are: SN90_ASO01 (1) SN90_ASI02 (2) SN90_ASI03 (3) SN90_DSM03 (4) SN90_DSM05_O (5) SN90_DSM04 (6) SN90_DSO01 (7) SN90_DSO02 (8) SN90_DSO03 (9) SN90_DSO04 (10) SN90_DSI01 (11) SN90_DSI02 (12) SN90_CIS02 (13) SN90_QRS02 (14) SN90_FCS01 (15) SN90_HSS01 (16) SN90_DSM05_I (17) SN90_VIRT_IO (18) SN90_DSM05_IO (19) SN90_DSM05_OI (20) SN90_FBS01 (21) SN90_S0E (22) SN90_RIO02 (23)
virtual_update_time	short	For virtual I/O modules only. Number of milliseconds between MFP module updates of function block values to the USM module.

sn90_establish_USM_remote_iomodule() *(continued)*

point and module initialization

RETURNS:

SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS:

This function must be issued for each remotely mounted simulated, actual, or virtual I/O module used by the simulation.

The IMDSM05 module can be one of four types:

- SN90DSM05_O (16 output channels).
- SN90DSM05_I (16 input channels).
- SN90DSM05_IO (eight input channels (zero through seven) and eight output channels (eight through 15)).
- SN90DSM05_OI (eight output channels (zero through seven) and eight input channels (eight through 15)).

The IMDSM04 is an I/O module that monitors a digital signal (e.g., a frequency or pulse) but creates an analog value that describes the signal. The modeling program must generate the analog value rather than the digital signal.

sn90_establish_USM_remote_point()

point and module initialization

PURPOSE: Defines a point (from a locally mounted I/O module) whose value is exchanged over the simulation network.

FORMAT: **short sn90_establish_USM_remote_point(***task_ID*, *address*, *iomodule_channel_number*, *point_name*, *iomodule_default_status*, *point_type_ethernet*, *IO_direction*, *IO_interval*, *block_number***)**

Parameter	Type	Description
task_ID	char*	Name of the USM module that contains this point. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
address	struct Slave Address	Address of the I/O module. Refer to ***REMOTE I/O MODULE SIMULATION*** in Section 4 for a description of the SlaveAddress structure.
iomodule_channel_number	short	Channel address corresponding to the function code for the particular I/O module type. Available addresses are dependent on the I/O module type. Refer to Table 5-3.
point_name	char*	Name of the I/O point. Must be a null terminated string. Maximum length of 32 alphanumeric characters.
iomodule_default_status	short	Output value of the USM module if the MFP module becomes unavailable. For I/O modules that do not get their default status from dipswitch settings, specify SN90_NA. Available options are: SN90_LOW (0) SN90_HIGH (1) SN90_HOLD (2) SN90_NA (3)
point_type_ethernet	short	Type of point. Available options are: SN90_DIGITAL (1) SN90_ANALOG (4)
IO_direction	short	Direction of data transmission relative to the MFP module. Available options are: SN90_RECV (0) SN90_SEND (1)
IO_interval	long	Time in milliseconds in which the point values should be exchanged between tasks. Minimum supported I/O interval is 250 msecs.
block_number	short	For virtual I/O module points only. MFP block number where tasks retrieve values.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: All points must be established for an I/O module even if it is not connected to a process model. For example, I/O modules having eight channels must establish all eight points on the USM module. The unused channels are given dummy point names.

sn90_establish_USM_remote_point() *(continued)*

point and module initialization

Default values are used only if communications with the MFP module are interrupted. The establishment of a malfunction on an I/O module or a channel will cause the output of that channel to remain locked at its last known value.

The iomodule_default_status needs to be specified for output channels for the following iomodule_type:

- SN90_CIS02.
- SN90_DSM05_IO.
- SN90_DSM05_O.
- SN90_DSM05_OI.
- SN90_HSS01.
- SN90_QRS02.
- SN90_VIRT_IO.

All other channels should use SN90_NA as the default status.

NOTES:

1. Default values for IMCIS02 and IMQRS02 I/O modules only apply to their analog output points.

2. Digital output defaults are low state (de-energized).

The IMDSM04 is an I/O module that monitors a digital signal (e.g., a frequency or pulse) but creates an analog value that describes the signal. The modeling program must generate the analog value rather than the digital signal.

sn90_establish_USM_remote_point() *(continued)*

point and module initialization

Table 5-5. I/O Module Channel Addresses

I/O Module Type	Channel Type	Available Addresses
SN90_ASI02	Analog input	0 - 14
SN90_ASI03	Analog input	0 - 15
SN90_ASO01	Analog output	0 - 13
SN90_CIS02	Analog input Analog output Digital input Digital output	0 - 3 4 - 5 6 - 8 9 - 12
SN90_DSI01	Digital input	0 - 15
SN90_DSI02	Digital input	0 - 15
SN90_DSM03	Digital input	0 - 15
SN90_DSM04	Analog input	0 - 7
SN90_DSM05_I	Digital input	0 - 15
SN90_DSM05_IO	Digital input Digital output	0 - 7 8 - 15
SN90_DSM05_O	Digital output	0 - 15
SN90_DSM05_OI	Digital output Digital input	0 - 7 8 - 15
SN90_DSO01	Digital output	0 - 7
SN90_DSO02	Digital output	0 - 7
SN90_DSO03	Digital output	0 - 7
SN90_DSO04	Digital output	0 - 15
SN90_FBS01	Analog input	0-14
SN90_FCS01	Analog input	0
SN90_HSS01	Analog output	0
SN90_QRS02	Analog input Analog output Digital input Digital output	0 - 3 4 - 5 6 - 8 9 - 12
SN90_SOE	Digital input	0-15
SN90_RIO02	Remote I/O	None
SN90_VIRT_IO	Analog output Digital output	0 - 7 8 - 15

sn90_freeze()

simulation control

PURPOSE: Puts the simulation network into a mode (freeze mode) that allows tasks to communicate notification messages, but control modules do not execute their control algorithms and simulations do not execute their process models. Also, USM modules stop transmitting I/O values and simulation time stops while the simulation network is in freeze mode.

FORMAT: **short sn90_freeze()**

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: SN90_FREEZE_ACK is provided by the **sn90_get_message()** function to the task issuing this function. The task should not issue the **sn90_update_values()** function while in freeze mode. The simulation manager sends the SN90_FREEZE_ACK notification message to the application task when it has received this function request. The simulation manager sends the SN90_FREEZE_DONE notification message upon successful completion of this function.

sn90_freeze_ack()

simulation control

PURPOSE: Notifies the simulation manager that the simulation task issuing this function has changed to freeze mode.

FORMAT: **short sn90_freeze_ack(*status*)**

Parameter	Type	Description
status	short	The completion status of the change to freeze mode. Possible statuses are: SN90_ERROR SN90_NORMAL

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

sn90_get_message()

messages

PURPOSE: Acquires the next available notification message from the simulation network.

FORMAT: **short sn90_get_message(***message_code***,** *message_status***)**

Parameter	Type	Description
message_code	short*	Memory address at which the type of message that was received is to be stored. Possible values that will be stored in this location include: SN90_CHANNEL_MALFUNCTIONS_COLLECTED SN90_END_OF_SIMULATION SN90_END_OF_SIMULATION_DONE SN90_FREEZE SN90_FREEZE_ACK SN90_FREEZE_DONE SN90_IOMOD_MALFUNCTIONS_COLLECTED SN90_MALFUNCTION_ACK SN90_NO_MESSAGE SN90_OVERRIDE_ACK SN90_OVERRIDES_COLLECTED SN90_POINT_NAMES_COLLECTED SN90_READY_TO_RUN SN90_RUN SN90_RUN_ACK SN90_RUN_DONE SN90_SIMULATION_TIME SN90_SIMULID_COLLECTED SN90_SNAPSHOT_COLLECTED SN90_SNAPSHOT_DELETE SN90_SNAPSHOT_DELETE_ACK SN90_SNAPSHOT_RESTORE SN90_SNAPSHOT_RESTORE_ACK SN90_SNAPSHOT_RESTORE_DONE SN90_SNAPSHOT_SAVE SN90_SNAPSHOT_SAVE_ACK SN90_SNAPSHOT_SAVE_DONE SN90_TASK_STATUS_COLLECTED
message_status	short*	Memory address at which the status code from the received message is to be stored. Possible values that will be stored in this location include: SN90_ERROR SN90_INV_REQ_CODE SN90_MEMORY_FAIL SN90_NORMAL SN90_NOT_REGISTERED SN90_RECEIVE_MSG_ERROR

sn90_get_message() *(continued)*

messages

RETURNS:

SN90_ERROR
SN90_NORMAL

REMARKS:

This function retrieves process command type messages. Point I/O is processed using the **sn90_update_values()** function.

SN90_NO_MESSAGE is defined to be zero. Therefore, the logical value of the message code will be TRUE if a message was received, and FALSE if no message was received. Similarly, SN90_NORMAL is defined to be zero. The logical value of the message status will be TRUE if an error occurred and FALSE if no error occurred.

The SN90_READY_TO_RUN status field contains the number of tasks ready to perform input/output.

sn90_get_run()

simulation control

PURPOSE: Unpacks an SN90_RUN notification message and returns the execution rate that was specified in the last call to **sn90_run()**.

FORMAT: **short sn90_get_run(***execution_rate***)**

Parameter	Type	Description
execution_rate	float*	Memory address at which the execution rate is to be stored.

RETURNS: SN90_INV_REQ_CODE
SN90_NORMAL

sn90_get_simulation_time()

simulation time management

PURPOSE: Unpacks an SN90_SIMULATION_TIME notification message and returns the simulation time parameters.

FORMAT: **short sn90_get_simulation_time(***second***,** *minute***,** *hour***,** *day***,** *month***,** *year***)**

Parameter	Type	Description
second	short*	Memory address at which the seconds specified in the **sn90_set_simulation_time()** function is to be stored.
minute	short*	Memory address at which the minutes specified in the **sn90_set_simulation_time()** function is to be stored.
hour	short*	Memory address at which the hours specified in the **sn90_set_simulation_time()** function is to be stored.
day	short*	Memory address at which the day specified in the **sn90_set_simulation_time()** function is to be stored.
month	short*	Memory address at which the month specified in the **sn90_set_simulation_time()** function is to be stored.
year	short*	Memory address at which the year specified in the **sn90_set_simulation_time()** function is to be stored.

RETURNS: SN90_INV_REQ_CODE
SN90_NORMAL

REMARKS: Use this function when SN90_SIMULATION_TIME is received.

sn90_get_snapshot_delete()

snapshot data

PURPOSE: Unpacks an SN90_SNAPSHOT_DELETE notification message and returns the snapshot ID.

FORMAT: **short sn90_get_snapshot_delete(***snapshot_ID***)**

Parameter	Type	Description
snapshot_ID	char*	Memory address at which the snapshot ID specified in the **sn90_snapshot_delete()** function will be stored. The address must point to an area in memory with enough room for 12 alphanumeric characters and a null termination character.

RETURNS: SN90_INV_REQ_CODE
SN90_NORMAL

sn90_get_snapshot_restore()

snapshot data

PURPOSE: Unpacks an SN90_SNAPSHOT_RESTORE notification message and returns the snapshot ID.

FORMAT: **short sn90_get_snapshot_restore(***snapshot_ID***)**

Parameter	Type	Description
snapshot_ID	char*	Memory address at which the snapshot ID specified in the **sn90_snapshot_restore()** function will be stored. The address must point to an area in memory with enough room for 12 alphanumeric characters and a null termination character.

RETURNS: SN90_INV_REQ_CODE
SN90_NORMAL

sn90_get_snapshot_save()

snapshot data

PURPOSE: Unpacks an SN90_SNAPSHOT_SAVE notification message and returns the snapshot ID.

FORMAT: **short sn90_get_snapshot_save(***snapshot_ID***)**

Parameter	Type	Description
snapshot_ID	char*	Memory address at which the snapshot ID specified in the **sn90_snapshot_save()** function will be stored. The address must point to an area in memory with enough room for 12 alphanumeric characters and a null termination character.

RETURNS: SN90_INV_REQ_CODE
SN90_NORMAL

sn90_iomodule_malf_add()

INFI 90 OPEN malfunction simulation

PURPOSE: Activates an INFI 90 OPEN malfunction in a locally mounted I/O module.

FORMAT: **short sn90_iomodule_malf_add(***task_ID, Xbus_address, malfunction_code***)**

Parameter	Type	Description
task_ID	char*	Name of the task that will issue the malfunction. Must be a null terminated string.
Xbus_address	short	I/O expander bus address of the I/O module being put into malfunction.
malfunction_code	short	Type of malfunction. Available options are: SN90_NO_RESPONSE (1) SN90_CALIBRATION_ERROR (2) SN90_CHANNEL_FAILURE (3) SN90_COUNTER_OVERFLOW (4) SN90_BLOWN_FUSE (5)

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: Refer to Table 4-1 for a description of the malfunction codes that are valid for each I/O module type. A successful execution of this function results in an SN90_MALFUNCTION_ACK notification message indicating the simulation manager has received the request.

> **NOTE:** To add a malfunction for a specific channel in a locally or remotely mounted I/O module, use the **sn90_channel_malf_add()** function.

sn90_iomodule_malf_clear()

INFI 90 OPEN malfunction simulation

PURPOSE: Clears a malfunction in a locally mounted I/O module that was activated by **sn90_iomodule_malf_add()**.

FORMAT: **short sn90_iomodule_malf_clear(***task_ID***,** *Xbus_address***)**

Parameter	Type	Description
task_ID	char*	Name of the task containing the malfunction to be cleared. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
Xbus_address	short	I/O expander bus address of the I/O module directly affected by the malfunction. Available addresses are 0 through 63.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: A successful execution of this function results in an SN90_MALFUNCTION_ACK notification message indicating the simulation manager has received the request.

sn90_next_channel_malf()

simulation network querying

PURPOSE:

Returns the point name and type of malfunction of the first or next active channel malfunction collected by either of the following:

sn90_collect_channel_malf()
sn90_collect_remote_channel_malf()

FORMAT:

short sn90_next_channel_malf(*point_name***,** *malfunction_code***)**

Parameter	Type	Description
point_name	char*	Memory address at which the name of the point experiencing the simulated malfunction will be stored. Must be a null terminated string. Maximum length of 32 alphanumeric characters.
malfunction_code	short*	Memory address at which the type of the malfunction will be stored. The only malfunction type currently available for channels is SN90_CHANNEL_MALFUNCTION (1).

RETURNS:

SN90_EOF
SN90_NORMAL

REMARKS:

Issue this function after the SN90_CHANNEL_MALFUNCTIONS_COLLECTED notification message is received.

A return value of SN90_NORMAL indicates that malfunction data has been stored in the indicated locations. A return value of SN90_EOF indicates that no more malfunctions are available.

sn90_next_iomodule_malf()

simulation network querying

PURPOSE: Returns the task ID, I/O module address, and type of malfunction of the first or next active I/O module malfunction collected by:

sn90_collect_iomodule_malf()

FORMAT: **short sn90_next_iomodule_malf(***task_ID***,** *Xbus_address***,** *malfunction_code***)**

Parameter	Type	Description
task_ID	char*	Memory address at which the name of the USM task controlling the malfunctioning module will be stored. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
Xbus_address	short*	Memory address at which the I/O expander bus address of the malfunctioning module will be stored. Available addresses are 0 through 63.
malfunction_code	short*	Memory address at which the type of the malfunction will be stored. Possible return types are: SN90_NO_RESPONSE (1) SN90_CALIBRATION_ERROR (2) SN90_CHANNEL_FAILURE (3) SN90_COUNTER_OVERFLOW (4) SN90_BLOWN_FUSE (5)

RETURNS: SN90_EOF
SN90_NORMAL

REMARKS: Issue this function after the notification message SN90_IOMOD_MALFUNCTIONS_COLLECTED is received.

A return value of SN90_NORMAL indicates that malfunction data has been stored in the indicated locations. A return value of SN90_EOF indicates that no more malfunctions are available.

sn90_next_override()

simulation network querying

PURPOSE:

Returns the name, override value, actual value, and quality of the first or next point override collected by:

sn90_collect_overrides()

FORMAT:

short sn90_next_override(*point_name***,** *override_value***,** *actual_value***,** *quality_check***)**

Parameter	Type	Description
point_name	char*	Memory address at which the name of the overridden point will be stored. Must be a null terminated string. Maximum length of 32 alphanumeric characters.
override_value	double*	Memory address at which the override output value of the point will be stored.
actual_value	double*	Memory address at which the actual output value of the point will be stored.
quality_check	short*	Memory address at which the quality control value specified when the override was established by **sn90_override_add()** will be stored. Possible quality check statuses are: SN90_NOQUALITY (0) SN90_QUALITY (1)

RETURNS:

SN90_EOF
SN90_NORMAL

REMARKS:

Issue this function after the SN90_OVERRIDES_COLLECTED notification message is received.

A return value of SN90_NORMAL indicates that override data has been stored in the indicated locations. A return value of SN90_EOF indicates that no more overrides are available.

sn90_next_pointname()

simulation network querying

PURPOSE:

Returns the task ID, name, type, I/O direction, and I/O interval for the first or next point name collected by:

sn90_collect_pointnames()

FORMAT:

short sn90_next_pointname(task_ID, point_name, point_type, IO_direction, IO_interval)

Parameter	Type	Description
task_ID	char*	Memory address at which the name of the task controlling the given point will be stored. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
point_name	char*	Memory address at which the point's name will be stored. Must be a null terminated string. Maximum length of 32 alphanumeric characters.
point_type	short*	Memory address at which the type of point will be stored. Possible return types are: SN90_BYTE (1) (1 byte) SN90_SHORT (2) (2 bytes) SN90_LONG (3) (4 bytes) SN90_FLOAT (4) (4 bytes) SN90_DOUBLE (5) (8 bytes)
IO_direction	short*	Memory address at which the direction of data transmission for the point relative to the task will be stored. Possible return directions are: SN90_RECV (0) SN90_SEND (1)
IO_interval	long*	Memory address at which the time interval (in milliseconds) between data transmissions for the point will be stored.

RETURNS:

SN90_EOF
SN90_NORMAL

REMARKS:

Issue this function after the SN90_POINT_NAMES_COLLECTED notification message is received.

The quality flag (SN90_QUALITY or SN90_NOQUALITY) affects only the feedback value of analog points being driven by analog output values from CIS or QRS modules (function code 79 blocks). If SN90_NOQUALITY is set, the feedback value will be in bad quality unless the override value matches the actual value of the point.

A return value of SN90_NORMAL indicates that point data has been stored in the indicated locations. A return value of SN90_EOF indicates that no more points are available.

sn90_next_remote_iomodule_malf()

simulation network querying

PURPOSE:

Returns the task ID, I/O module address, and type of malfunction of the first or next active I/O module malfunction collected by:

sn90_collect_remote_iomodule_malf()

FORMAT:

short sn90_next_remote_iomodule_malf(*task_ID***,** *address***,** *malfunction_code***)**

Parameter	Type	Description
task_ID	char*	Memory address at which the name of the USM task controlling the malfunctioning module will be stored. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
address	struct Slave Address*	Memory address at which the address structure of the malfunctioning module will be stored.
malfunction_code	short*	Memory address at which the type of the malfunction will be stored. Possible return types are: SN90_NO_RESPONSE (1) SN90_CALIBRATION_ERROR (2) SN90_CHANNEL_FAILURE (3) SN90_COUNTER_OVERFLOW (4) SN90_BLOWN_FUSE (5)

RETURNS:

SN90_EOF
SN90_NORMAL

REMARKS:

Issue this function after the notification message SN90_IOMOD_MALFUNCTIONS_COLLECTED is received.

A return value of SN90_NORMAL indicates that malfunction data has been stored in the indicated locations. A return value of SN90_EOF indicates that no more malfunctions are available.

A remotely mounted I/O module controlled by a Remote Master Processor module that is in malfunction will report an SN90_NO_RESPONSE malfunction, even if a different malfunction was in effect before the RMP module went into malfunction. When the RMP module's malfunction is cleared, remotely mounted I/O modules controlled by that RMP module will report any malfunctions that were in effect before the RMP module's malfunction was established.

sn90_next_simulation_ID()

simulation network querying

PURPOSE: Returns the ID of the first or next simulation collected by:

sn90_collect_simulation_ID()

FORMAT: **short sn90_next_simulation_ID(***simulation_ID***)**

Parameter	Type	Description
simulation_ID	char*	Memory address at which the simulation ID will be stored. Must be a null terminated string. Maximum length of 16 alphanumeric characters.

RETURNS:
SN90_EOF
SN90_NORMAL

REMARKS: There is no notification message generated by the **sn90_collect_simulation_ID()** function that would indicate the list of simulation IDs is available. Call this function immediately after the successful execution of the **sn90_collect_simulation_ID()** function.

A return value of SN90_NORMAL indicates that a simulation identifier has been stored in the indicated locations. A return value of SN90_EOF indicates that no more simulation identifiers are available.

sn90_next_snapshot_task()

simulation network querying

PURPOSE:

Returns the task ID and snapshot save status of the first or next snapshot task collected by:

sn90_collect_snapshot()

FORMAT:

short sn90_next_snapshot_task(*task_ID***,** *snap_status***)**

Parameter	Type	Description
task_ID	char*	Memory address at which the name of the task is to be stored. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
snap_status	int*	Memory address at which the save status of the task is to be stored. Possible save statuses are: SN90_SNAP_OK (0) SN90_SNAP_TIMEOUT (1) SN90_SNAP_FILE_ERROR (2) SN90_SNAP_COMM_ERROR (3) SN90_SNAP_REFUSED (4) SN90_SNAP_MFP_ERROR (5)

RETURNS:

SN90_EOF
SN90_NORMAL

REMARKS:

Issue this function after the SN90_SNAPSHOT_COLLECTED notification message is received.

A return value of SN90_NORMAL indicates that snapshot save data for a task has been stored in the indicated locations. A return value of SN90_EOF indicates that no more data is available.

sn90_next_task_status()

simulation network querying

PURPOSE:

Returns the task ID and run, MFP module, communication, restore, and save status of the first or next task status collected by:

sn90_collect_task_status()

FORMAT:

short sn90_next_task_status(*task_ID***,** *run_status***,** *MFP_status***,** *comm_status***,** *restore_status***,** *save_status***)**

Parameter	Type	Description
task_ID	char*	Memory address at which the name of the task is to be stored. Must be a null terminated string. Maximum length of 16 alphanumeric characters.
run_status	short*	Pointer to the run status of the task. Possible statuses are: SN90_FREEZE (106) SN90_FREEZE_FAILED (3) SN90_RUN (105) SN90_RUN_FAILED (4) SN90_UNKNOWN (127)
MFP_status	short*	Memory address at which the MFP status of the task is to be stored. If the task contains no MFP modules, the SN90_NOTMFP status is returned. Possible MFP statuses are: SN90_CONFIGURE (0) SN90_EXECUTE (3) SN90_FAILOVER (1) SN90_MFP_ERROR (2) SN90_NOTMFP (4) SN90_UNKNOWN (127)
comm_status	short*	Memory address at which the task's communication status is to be stored. Possible communication statuses are: SN90_ERROR (99) SN90_NORMAL (0)
restore_status	short*	Memory address at which the status of the last attempt to restore data for this task is to be stored. Possible restore statuses are: SN90_ERROR (99) SN90_NORMAL (0)
save_status	short*	Memory address at which the status of the last attempt to save data for this task is to be stored. Possible save statuses are: SN90_ERROR (99) SN90_NORMAL (0)

RETURNS:

SN90_EOF
SN90_NORMAL

sn90_next_task_status() *(continued)*

simulation network querying

REMARKS:

Issue this function after the SN90_TASK_STATUS_COLLECTED notification message is received.

A return value of SN90_NORMAL indicates that data for a task has been stored in the indicated locations. A return value of SN90_EOF indicates that no more data is available.

sn90_open()

task control

PURPOSE: Notifies the task manager of the name of this task.

FORMAT: **short sn90_open(***task_ID***)**

Parameter	Type	Description
task_ID	char*	Name of the task. Must be a null terminated string. Maximum length of 16 alphanumeric characters.

RETURNS:
SN90_CONNECT_ERROR
SN90_MEMORY_FAIL
SN90_NORMAL

sn90_override_add()

point I/O values

PURPOSE:

Causes the actual output value of a point to be overridden with the specified value.

FORMAT:

short sn90_override_add(*point_name***,** *override_value***,** *quality_check***)**

Parameter	Type	Description
point_name	char*	Name of the point to be overridden. Must be a null terminated string. Maximum length of 32 alphanumeric characters.
override_value	double	The point output value being substituted for the actual output value.
quality_check	short	A flag that tells the MFP module to either enable or disable the quality check on the feedback value of analog points. Possible quality options are: SN90_NOQUALITY SN90_QUALITY

RETURNS:

SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS:

The actual output value is overridden externally to the task and the sending task is not aware that a point output value is being overridden.

The quality flag (SN90_QUALITY or SN90_NOQUALITY) affects only the feedback value of analog points being driven by analog output values from CIS or QRS modules (function code 79 blocks). If SN90_NOQUALITY is set, the feedback value will be in bad quality unless the override value matches the actual value of the point.

A successful execution of this function results in an SN90_OVERRIDE_ACK notification message indicating the simulation manager has received the request.

sn90_override_clear()

point I/O values

PURPOSE: Clears an active override for a specific point.

FORMAT: **short sn90_override_clear(***point_name***)**

Parameter	Type	Description
point_name	char*	Name of the point from which the override is being removed. Must be a null terminated string. Maximum length of 32 alphanumeric characters.

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: A successful execution of this function results in an SN90_OVERRIDE_ACK notification message indicating the simulation manager has received the request.

sn90_read_value()

point I/O values

PURPOSE:

Provides the value of the specified point from an area accessible to the API services.

FORMAT:

short sn90_read_value(*point_handle***,** *point_value***)**

Parameter	Type	Description
point_handle	SN90_ POINT*	Memory address of a data structure used to store information about a point.
point_value	double	Memory address at which the value of the point is to be stored.

RETURNS:

SN90_ERROR
SN90_NORMAL

REMARKS:

If the points are established (using the **sn90_establish_point()** function) by supplying the address of the point value, then this function does not need to be issued. The point_handle data structure must be allocated persistent memory storage space.

sn90_register()

task control

PURPOSE: Registers the task using this function with the specified simulation (i.e., simulation manager).

FORMAT: **short sn90_register(***simulation_ID***)**

Parameter	Type	Description
simulation_ID	char*	Name of the simulation that the task issuing this function will be part of. Must be a null terminated string. Maximum length of 16 alphanumeric characters.

RETURNS:
SN90_CONNECT_ERROR
SN90_INVALID_SIMULID
SN90_NORMAL
SN90_TIMEOUT

REMARKS: Tasks must be registered with simulations that use them. This function must be the second API function the task issues (**sn90_open()** is first). This function will fail if the task manager is not running or if the requested simulation manager is not running. The simulation ID must match the simulation_ID used in starting the corresponding simulation manager task.

sn90_remote_iomodule_malf_add()

INFI 90 OPEN malfunction simulation

PURPOSE: Activates an INFI 90 OPEN malfunction in a remotely mounted I/O module.

FORMAT: **short sn90_remote_iomodule_malf_add(***task_ID***,** *address***,** *malfunction_code***)**

Parameter	Type	Description
task_ID	char*	Name of the task that will issue the malfunction. Must be a null terminated string.
address	struct Slave Address	Address of the I/O module being put into malfunction.
malfunction_code	short	Type of malfunction. Available options are: SN90_NO_RESPONSE (1) SN90_CALIBRATION_ERROR (2) SN90_CHANNEL_FAILURE (3) SN90_COUNTER_OVERFLOW (4) SN90_BLOWN_FUSE (5)

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: Refer to Table 4-1 for a description of the malfunction codes that are valid for each I/O module type. A successful execution of this function results in an SN90_MALFUNCTION_ACK notification message indicating the simulation manager has received the request.

> **NOTE:** To add a malfunction for a specific channel in a locally or remotely mounted I/O module, use the **sn90_channel_malf_add()** function.

Placing an IMRIO02 module used as a Remote Master Processor into malfunction will cause all channels of all I/O modules controlled by that RMP module to report SN90_NO_RESPONSE malfunctions. Any previously existing malfunctions in remote I/O modules will be temporarily overridden, but they will be restored when the RMP module's malfunction is cleared.

A locally mounted I/O module can be placed into malfunction using this function by setting the localAddress field of the SlaveAddress structure to the I/O module's expander bus address and setting the remoteNode field to -1.

sn90_run()

simulation control

PURPOSE: Directs all tasks within the same simulation to go to run mode.

FORMAT: **short sn90_run(***execution_rate***)**

Parameter	Type	Description
execution_rate	float	How fast, relative to real time, the simulation runs. The range of allowable execution rates is 0.001 through 50. Refer to the discussion below for guidelines on setting the execution rate.

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: The maximum execution rate setting for any simulation depends on the dynamics of the simulation. Complex, rapidly changing simulations may function properly when set as high as five. Simple, slowly changing simulations may function properly when set as high as ten.

Changing the run rate does not change the frequency of I/O exchange between the USM module and process simulation, nor does it change the frequency of control logic execution. The MFP module will adjust the elapsed time by the run rate in any time dependent calculations.

The simulation manager sends the SN90_RUN_ACK notification message to the application task when it has received this function request. The simulation manager sends the SN90_RUN_DONE notification message upon successful execution of this function.

sn90_run_ack()

simulation control

PURPOSE: Notifies the simulation manager task that the task has changed to run mode.

FORMAT: **short sn90_run_ack(*status*)**

Parameter	Type	Description
status	short	Completion status of the change to run mode. Possible statuses are: SN90_ERROR SN90_NORMAL

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

sn90_set_simulation_time()

simulation time management

PURPOSE: Requests all tasks within the simulation to use this time.

FORMAT: **short sn90_set_simulation_time(***second***,** *minute***,** *hour***,** *day***,** *month***,** *year***)**

Parameter	Type	Description
second	short	The second to which the simulation time will be set.
minute	short	The minute to which the simulation time will be set.
hour	short	The hour to which the simulation time will be set.
day	short	The day to which the simulation time will be set.
month	short	The month to which the simulation time will be set.
year	short	The year to which the simulation time will be set.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: The MFP module clock is set to the requested simulation time.

sn90_simulation_time_ack()

simulation time management

PURPOSE: Notifies the simulation manager task that the task has received an SN90_SIMULATION_TIME notification message code.

FORMAT: **short sn90_simulation_time_ack(*status*)**

Parameter	Type	Description
status	short	Completion status of the **sn90_set_simulation_time()** function. Possible statuses are: SN90_ERROR SN90_NORMAL

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

sn90_snapshot_delete()

snapshot data

PURPOSE: Deletes all data for the specified snapshot and instructs client programs to do the same.

FORMAT: **short sn90_snapshot_delete(***snapshot_ID*, *snapshot_directory***)**

Parameter	Type	Description
snapshot_ID	char*	Identification for the application programs to use to select the snapshot to delete. Must be a null terminated string. Maximum length of 12 alphanumeric characters.
snapshot_directory	char*	Name of the directory containing the snapshot to delete. Must be a null terminated string. Maximum length of 64 alphanumeric characters.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: The simulation manager sends the SN90_SNAPSHOT_DELETE_ACK notification message to the application task when it has received this function request. There is no notification when the snapshot delete operation is complete.

sn90_snapshot_delete_ack()

snapshot data

PURPOSE: Notifies the simulation manager that the task has completed a snapshot deletion operation.

FORMAT: **short sn90_snapshot_delete_ack(*status*)**

Parameter	Type	Description
status	short	Completion status of the snapshot deletion operation. Possible status are: SN90_ERROR SN90_NORMAL

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

sn90_snapshot_restore()

snapshot data

PURPOSE: Uses the dynamic data and static configuration information found in a snapshot file to restore all USM and MFP modules in a simulation to a previously saved condition.

FORMAT: **short sn90_snapshot_restore(***snapshot_ID***,** *snapshot_directory***)**

Parameter	Type	Description
snapshot_ID	char*	Identification for the application programs to use to select the desired snapshot. Must be a null terminated string. Maximum length of 12 alphanumeric characters.
snapshot_directory	char*	Name of the directory containing the snapshot files. Must be a null terminated string. Maximum length of 64 alphanumeric characters.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: The simulation manager sends the SN90_SNAPSHOT_RESTORE_ACK notification message to the application task when it has received this function request. The simulation manager sends the SN90_SNAPSHOT_RESTORE_DONE notification message upon successful execution of the function.

sn90_snapshot_restore_ack()

snapshot data

PURPOSE: Notifies the simulation manager that the task has restored its snapshot.

FORMAT: **short sn90_snapshot_restore_ack(*status*)**

Parameter	Type	Description
status	short	Completion status of the **sn90_snapshot_restore()** function. Possible statuses are: SN90_ERROR SN90_NORMAL

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

sn90_snapshot_restore_dynamic()

snapshot data

PURPOSE: Uses the dynamic data found in a snapshot file to restore all USM and MFP modules in a simulation to a previously saved condition.

FORMAT: **short sn90_snapshot_restore_dynamic(***snapshot_ID***,** *snapshot_directory***)**

Parameter	Type	Description
snapshot_ID	char*	Identification for the application programs to use to select the desired snapshot. Must be a null terminated string. Maximum length of 12 alphanumeric characters.
snapshot_directory	char*	Name of the directory containing the snapshot files. Must be a null terminated string. Maximum length of 64 alphanumeric characters.

RETURNS: SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: The dynamic restore will not restore tuning specifications to the MFP module. The simulation manager sends the SN90_SNAPSHOT_RESTORE_ACK notification message to the application task when it has received this function request. The simulation manager sends the SN90_SNAPSHOT_RESTORE_DONE notification message upon successful execution of the function.

If a dynamic restore operation is requested for a USM task whose MFP module contains C, batch or data files, then a full restoration, including tuning specifications, will be performed for that task. This is to ensure that C and batch programs will be in valid states when the simulation is unfrozen.

sn90_snapshot_restore_task()

snapshot data

PURPOSE:

Restores one task of a simulation to a previously saved condition. If the task is a USM task, static and dynamic data for the associated USM and MFP modules are both restored.

FORMAT:

short sn90_snapshot_restore_task(*snapshot_ID***,** *snapshot_directory***,** *task_ID***)**

Parameter	Type	Description
snapshot_ID	char*	Identification for the application programs to use to select the desired snapshot. Must be a null terminated string. Maximum length of 12 alphanumeric characters.
snapshot_directory	char*	Name of the directory containing the snapshot files. Must be a null terminated string. Maximum length of 64 alphanumeric characters.
task_ID	char*	Name of the task containing the USM and MFP module. Must be a null terminated string. Maximum length of 16 alphanumeric characters.

RETURNS:

SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS:

The simulation manager sends the SN90_SNAPSHOT_RESTORE_ACK notification message to the application task when it has received this function request. The simulation manager sends the SN90_SNAPSHOT_RESTORE_DONE notification message upon successful execution of the function.

sn90_snapshot_restore_task_dyn()

snapshot data

PURPOSE: Restores dynamic data for one task of a simulation to a previously saved condition.

FORMAT: **short sn90_snapshot_restore_task_dyn(***snapshot_ID***,** *snapshot_directory***,** *task_ID***)**

Parameter	Type	Description
snapshot_ID	char*	Identification for the application programs to use to select the desired snapshot. Must be a null terminated string. Maximum length of 12 alphanumeric characters.
snapshot_directory	char*	Name of the directory containing the snapshot files. Must be a null terminated string. Maximum length of 64 alphanumeric characters.
task_ID	char*	Name of the task containing the USM and MFP module. Must be a null terminated string. Maximum length of 16 alphanumeric characters.

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS: The dynamic restore will not restore tuning specifications to the MFP module. The simulation manager sends the SN90_SNAPSHOT_RESTORE_ACK notification message to the application task when it has received this function request. The simulation manager sends the SN90_SNAPSHOT_RESTORE_DONE notification message upon successful execution of the function.

> **NOTE:** Since there is no distinction between static and dynamic data for application task snapshots, **sn90_snapshot_restore_task_dyn()** is equivalent to **sn90_snapshot_restore_dynamic()** for application tasks.

sn90_snapshot_save()

snapshot data

PURPOSE:

Copies the dynamic data and static configuration information about all USM and MFP modules in a simulation to a snapshot file. Also sends save snapshot commands to all application tasks in the simulation.

FORMAT:

short sn90_snapshot_save(*snapshot_ID***,** *snapshot_directory***)**

Parameter	Type	Description
snapshot_ID	char*	Identification string to be applied to this snapshot. Must be a null terminated string. Maximum length of 12 alphanumeric characters.
snapshot_directory	char*	Name of the directory into which the snapshot data is to be stored. Must be a null terminated string. Maximum length of 64 alphanumeric characters.

RETURNS:

SN90_NORMAL
SN90_SEND_MSG_ERROR

REMARKS:

The simulation manager sends the SN90_SNAPSHOT_SAVE_ACK notification message to the application task when it has received this function request. The simulation manager sends the SN90_SNAPSHOT_SAVE_DONE notification message upon successful execution of the function.

NOTES:

1. If the requested directory does not exist, it will be created.

2. The snapshot ID is not used by the simulation manager. It is provided for possible use by application tasks.

sn90_snapshot_save_ack()

snapshot data

PURPOSE: Notifies the simulation manager task that the task has saved its snapshot.

FORMAT: **short sn90_snapshot_save_ack(*status*)**

Parameter	Type	Description
status	short	Completion status of the **sn90_snapshot_restore()** function. Possible statuses are: SN90_ERROR SN90_NORMAL

RETURNS:
SN90_NORMAL
SN90_SEND_MSG_ERROR

sn90_update_values()

point I/O values

PURPOSE: Causes the task to send output values it generates to their destinations and to receive any queued input values from the simulation network.

FORMAT: **short sn90_update_values()**

RETURNS:
SN90_CONNECT_ERROR
SN90_MEMORY_FAIL
SN90_NORMAL
SN90_NOT_CONNECTED
SN90_RECEIVE_MSG_ERROR
SN90_SEND_MSG_ERROR

REMARKS: If the task fails to issue this function, output values will not be sent out and input values will not be received. Only points that have exceeded the I/O interval since the last transmit time will actually be sent when this function is issued.

This function should be called as frequently as required to assure values are processed at their I/O interval. The copying of values from or to application memory occurs only during the execution of this function.

If an application uses the **sn90_read_value()** and **sn90_write_value()** functions, then an update sequence would be as follows:

sn90_write_value() for each send point.

sn90_update_values()

sn90_read_value() for each receive point.

It is recommended that the points be established so as not to require the use of **sn90_write_value()** and **sn90_read_value()** functions.

sn90_write_value()

point I/O values

PURPOSE: Passes a point value to the API services for forwarding to the destination task.

FORMAT: **short sn90_write_value(***point_handle***,** *point_value***)**

Parameter	Type	Description
point_handle	SN90_ POINT*	Memory address of a data structure used to store information about a point.
point_value	double	Value to which the point is being set.

RETURNS: SN90_ERROR

SN90_NORMAL

REMARKS: If the points are established supplying the address of the point value in the **sn90_establish_point()** function, then this function does not need to be issued.

SECTION 6 - OPERATING PROCEDURES

INTRODUCTION

This section discusses the steps involved in configuring and operating a simulation under normal conditions.

> **NOTE:** Refer to Appendix C for information on hardware setups that must be performed before operating a simulation. Failure to have the MFP and USM modules configured properly will result in simulation operation problems.

CONFIGURATION

Configuration File

When the simulation manager program starts, it checks to see if a file named ***sm.cfg*** exists in the current directory. If the file exists, the simulation manager program reads it and adjusts any parameters listed in that file accordingly.

The ***sm.cfg*** file is designed to enable users to match time-out, delay, and other values to their individual systems without needing to rebuild the programs. During the installation process, a sample version of the ***sm.cfg*** file was written into the same directory that the ***sn90sm*** executable file was written. (By default, this would have been ***/sn90/bin***.)

The pound sign (#) is a comment delimiter in the ***sm.cfg*** file. Anything appearing on a line after a pound sign is ignored. In the sample ***sm.cfg*** file, all parameters are preceded by pound signs, meaning that all values in that file will be ignored and default values will be used. For reference, the sample file contains default values for all of the parameters.

To adjust a parameter in the ***sm.cfg*** file, remove the pound sign from the line containing the parameter, and adjust the value for that parameter as desired. The change will take effect the next time the simulation manager is started.

Configurable Parameters

Table 6-1 shows the parameters that can be configured using the ***sm.cfg*** file:

Table 6-1. Configurable Parameters

Parameter	Default	Description
HowAreYouTimeout	60.0 seconds	Amount of time between how-are-you messages sent to the USMs. If a USM fails to reply to a how-are-you message, the simulation manager assumes the connection has been broken for some reason. In that case, it will automatically attempt to reestablish the connection.

Table 6-1. Configurable Parameters (Continued)

Parameter	Default	Description
HardKeyTimeout	3600.0 seconds (one hour)	Average amount of time between checks for the presence of the hardware key on the computer running the Simulation Manager. The actual time between checks can vary between one minute and twice the time specified in this parameter. **NOTE:** If this parameter is set greater than 3600.0, it will be reduced to 3600.0.
SaveTimeOut	90.0 seconds	Amount of time allocated for completion of an entire snapshot save operation.
RestoreTimeOut	240.0 seconds	Amount of time allocated for completion of an entire snapshot restore operation.
MalfunctionAckTimeot	20.0 seconds)	Amount of time allowed for a malfunction command to be acknowledged.
OverrideAckTimeout	20.0 seconds	Amount of time allowed for an override add or override clear command to be acknowledged.
ActiveOverridesTimeout	40.0 seconds	Amount of time allowed for a task to generate a list of its active overrides.
RoutingTableAckTimeout	20.0 seconds	Amount of time allowed for a USM to acknowledge receipt of a set of routing table messages.
PointListAckTimeout	20.0 seconds	Amount of time allowed for an application task to acknowledge receipt of a set of point list messages.
IOgroupAckTimeout	40.0 seconds	Amount of time allowed for a USM to acknowledge a set of I/O group messages.
collectOverrideTimeout	40.0 seconds	Amount of time allowed for all overrides to be collected in response to an **sn90_collect_overrides()** function call.
Client_runTimeout	60.0 seconds	Amount of time allowed for an application task to respond to a run command.
Client_freezeTimeout	90.0 seconds	Amount of time allowed for an application task to respond to a freeze command.
Client_saveTimeout	60.0 seconds	Amount of time allowed for an application task to respond to a save command.
Client_restoreTimeout	60.0 seconds	Amount of time allowed for an application task to respond to a restore command.
Client_endSimTimeout	60.0 seconds	Amount of time allowed for an application task to respond to a end simulation command.
USM_runTimeout	60.0 seconds	Amount of time allowed for a USM task to respond to a run command.
USM_freezeTimeout	60.0 seconds	Amount of time allowed for a USM task to respond to a freeze command.
USM_saveTimeout	120.0 seconds	Amount of time allowed for a USM task to retrieve static data from its USM module during a snapshot save operation.
USM_saveDynamicTimeout	120.0 seconds	Amount of time allowed for a USM task to retrieve dynamic data from the USM module during a snapshot save operation.

Table 6-1. Configurable Parameters (Continued)

Parameter	Default	Description
USM_restoreTimeout	300.0 seconds	Amount of time allowed for restoration of data to a single USM/MFP module pair.
USM_endSimTimeout	300.0 seconds	Amount of time allowed for a USM to acknowledge completion of tasks performed in response to an end simulation command.
USM_waitDynamicDataAvailTimeout	720.0 seconds	Amount of time allowed for a USM to collect dynamic data from its MFP module and inform the simulation manager that the data has been collected.
USM_PointTableAckTimeout	100.0 seconds	Amount of time allowed for a USM task to acknowledge receipt of a list of simulated points.
USM_SlaveMapAckTimeout	100.0 seconds	Amount of time allowed for a USM task to acknowledge receipt of the map of simulated I/O modules.
USM_waitFilesTimeout	20.0 seconds	Amount of time allowed for a USM task to retrieve data files from its USM during a snapshot save operation.
PointTableDelay	0.0 seconds	Amount of time to wait between point table messages sent to a USM module to avoid overflowing the module's receive buffer. This will need to be adjusted only if very fast network cards are being used.
SlaveMapDelay	0.0 seconds	Amount of time to wait between I/O module map messages sent to a USM module to avoid overflowing the module's receive buffer. This will need to be adjusted only if very fast network cards are being used.
RestoreDelay	0.0 seconds	Amount of time to wait between snapshot data messages sent to a USM module to avoid overflowing the module's receive buffer. This will need to be adjusted only if very fast network cards are being used.
RestoreBlast	1	Number of restore messages to send to a USM before a delay to allow the USM to process data it has already received. This will need to be adjusted only if very fast network cards are being used.
USM_watchDogCount	10	Number of watchdog intervals that a USM should wait before assuming that its connection to the simulation manager has been lost.
USM_watchDogInterval	30 seconds	Length of each watchdog interval.
USM_LAN_receive_buffer	16 Kbytes for OS/2 systems, 48 Kbytes for all others	Size of the USM task's receive buffer. **NOTE:** This value must never be greater than 16 Kbytes on OS/2 machines.
StartupTime	300 seconds	Time between the start of the simulation manager program and the start of checking to make sure that all USMs have been completely initialized.
USM_LAN_send_buffer	16 Kbytes	Size of the USM task's send buffer. **NOTE:** This value must never be greater than 16 Kbytes on OS/2 machines.

Table 6-1. Configurable Parameters (Continued)

Parameter	Default	Description
Bins	10	Number of USM broadcast groups.
IngnoreRankingTableAckFailure	0	Determines whether or not to ignore acknowledge failure. **NOTE:** Do not change to 1 unless directed by Elsag Bailey.
QueueMessages	1	Determines whether or not to queue snapshot data.
StaticSaveAckTimeout	90.0	Amount of time allowed for a USM to respond to a static save request.
DelayVFI	0.0	Delay before requesting VFI.

SIMULATION OPERATION

The following steps are required to start up, run, and shut down a simulation:

1. Run the ***sn90tm*** program (the task manager) on all computers that will be running tasks that will communicate with the simulation. Commonly, this program is run with debug option 253 using the command:

sn90tm -D 253 ENTER

This asks the task manager to report all of the tasks that it knows about once a second. Refer to ***DEBUGGING COMMAND LINE OPTIONS*** for information on the task manager command line debugging options.

NOTE: Insure that only one task manager is running on a simulation computer at a time.

2. Run the ***sn90sm*** program (the simulation manager) on one computer in the network. A name must be given to the simulation in this command. Also, this program is usually run with debug option 2 using the command:

sn90sm *simname* **-D 2** ENTER

Refer to ***DEBUGGING COMMAND LINE OPTIONS*** for information on the simulation manager command line debugging options. Refer to ***SIMULATION MANAGER COMMAND LINE*** for complete information on all simulation manager command line options.

3. Use the following functions to tell the simulation manager and the USM modules what I/O modules and points are going to be simulated.

sn90_establish_USM()
sn90_establish_USM_iomodule()
sn90_establish_USM_point
sn90_establish_USM_remote_iomodule()
sn90_establish-USM_remote_point()

Refer to Section 5 for information on these functions.

4. Issue the **sn90_establish_USM_done()** function after all USMs, I/O modules, and points have been defined.

A dedicated application program such as the example program ***USMconfig*** can be used to establish the USMs, I/O modules, and points, or these functions can be incorporated into process modeling programs. Typically, a dedicated program is used, with modeling programs being started after the USM configuration process is complete, but this is not required.

5. Start the process modeling programs. Every modeling program must first call sn90_open() to connect with the task manager, and then call sn90_register() to establish communications with the simulation manager. The program must then call the **sn90_establish_point()** function for every value that it will be receiving from or sending to the simulation manager or to other modeling programs in the network. After the program has established all of its points, the **sn90_establish_point_done()** function must be called.

6. After the *Simulation startup complete* message is written to the log file by the simulation manager, begin the simulation by issuing the **sn90_run()** function.

7. Before shutting down the simulation, issue the **sn90_freeze()** function. This is not required, but it will prevent *NO TX* error messages from appearing in the log file in response to the end simulation command.

8. To shut down the simulation, issue the **sn90_end_simulation()** function.

There is no restriction on using any simulation functions in any application programs. The simulation application programming interface does not distinguish between configuration, control, and modeling programs.

SIMULATION MANAGER COMMAND LINE

Several command line options are available for the simulation manager program. Most command line options are flagged by a dash followed by a single character. If any further arguments are needed for an option, a space is required between the option and the argument. Command line options are case

sensitive. Option characters may not be combined. If an option is not specified, then a default is used.

The full simulation manager command line syntax is:

sn90sm *simname* [**-d** *staticdir*] [**-D** *debugflags*] [**-f** *outfile*] [**-g** *maxpoints*] [**-keepalive**] [**-l** *logdir*] [**-p** *serport*] [**-q**]

where:

simname — The simulation identifier, maximum of 16 alphanumeric characters.

-d *staticdir* — Specifies the main snapshot directory. This is the name of the directory that will hold the main snapshot information. This directory must exist before starting up the simulation. The default is a sub-directory named ***snapDir*** off the current directory.

The main snapshot directory holds files of static data (block numbers, function codes, and specifications) from every MFP that is used in the simulation. If a block is added or deleted or a specification is tuned, the change is recorded in the appropriate snapshot file in this directory. Also, certain operator actions such as changing a station's mode from manual to automatic or vice versa cause changes that are recorded here. When a snapshot is saved, the static data stored in this directory is copied into the directory specified in the **sn90_snapshot_save()** function parameters, rather than taking the time to download the data again from the MFP.

-D *debugflags* — Specifies the debugging information to be displayed. The default is 0 (no debugging messages). Refer to ***DEBUGGING COMMAND LINE OPTIONS*** for a description of all debugging flags.

-f *outfile* — Specifies the output file. This is the name of the file that will receive output from the simulation manager. The default is no output file.

> **NOTE:** The output file is distinct from the log file, although if the D_PRINT_LOG debug flag (refer to Table 6-2) is on, log messages will also appear in the output file.

-g *maxpoints*	Specifies the maximum number of points in an I/O group. The default is 300.
-keepalive	Do not stop the simulation manager after an end simulation command. The default is to stop the simulation manager after an end simulation command.

> **NOTE:** The count of USM modules that the simulation manager is connected to is not reset after an end simulation command with this option in effect. Therefore, if a new configuration is established, a message saying *Cannot add more USMs* and several messages giving I/O module addresses or point names and saying *Task not found* may appear in the log file. This problem will be fixed in a future release.

-l *logdir* *(lower case L)*	Specifies the log file directory. This is the name of the directory in which the log file is to be written. This directory must exist before starting up the simulation. The default is the current directory.
-p *serport*	Specifies the serial port on which hardware key is connected. Valid inputs are 1 and 2. The default is 0 (check both ports for a hardware key).
-q	Disables queuing of snapshot data.

Example 1: **sn90sm** *simname* **-d /sn90/main_snapshot** (specifies the snapshot directory as sn90/main_snapshot)

Example 2: **sn90sm** *simname* **-D 2** (displays messages that are written into the log file)

Example 3: **sn90** *simname* **-f /sn90/log/sn90.out** (specifies the output file as ***sn90.out***)

Example 4: **sn90sm** *simname* **-g 100** (limits I/O groups to 100 points each)

Example 5: **sn90sm** *simname* **-keepalive** (keeps the SM alive after an end command)

Example 6: **sn90sm** *simname* **-l /sn90/log** (specifies the log file directory /sn90/log)

Example 7: **sn90sm** *simname* **-p 1** (specifies to check serial port 1)

Example 8: **sn90sm** *simname* **-q** (disables queueing of snapshot data)

DEBUGGING COMMAND LINE OPTIONS

Debug flags can be defined by supplying the sum of selected items shown in Table 6-2 as a command line argument to the simulation manager. The **-D** *<debug flags>* can be supplied as an optional command line argument to activate various

classifications of debugging messages. For example, **-D 6** activates both D_PRINT_LOG and D_PACKETS.

NOTES:

1. For normal operation, the **-D 2** (print log message) option should be used so that the progress of the simulation manager can be observed in an output window.

2. Any other debug options should only be used at the instruction of Bailey engineers, since large quantities of debug messages can adversely affect the performance of the simulation manager.

3. The task manager can also accept debug flags. A value of **-D 253** is recommended. This will cause the simulation tasks to be listed every second. The task manager's default debug value is zero, which produces no output at all.

Table 6-2. Debugging Options

Flag Name	Hex Value	Dec Value	Message Class Description
D_BASE	0x0001	1	Base function call tracing
D_PRINT_LOG	0x0002	2	Log messages
D_PACKETS	0x0004	4	Packet message headers
D_USM	0x0008	8	USM related messages
D_MISC	0x0010	16	Miscellaneous messages
D_OBJECTS	0x0020	32	Object messages
D_WORK	0x0040	64	Work function related messages
D_SM	0x0080	128	SM control messages
D_BTREE	0x0100	256	B-tree debugging messages
D_API	0x0200	512	API related messages
D_HARDKEY	0x0400	1024	Software control hard-key related messages
D_SNAPSHOT	0x0800	2048	Snapshot debugging
D_POINT_DATA	0x1000	4096	Point table debugging
D_HOW_ARE_YOU	0x2000	8192	How-are-you protocol debugging
D_BUFFERS	0x4000	16384	Print entire message contents
D_REMOTEIO	0x8000	32768	Remote I/O debugging
D_ALL	0xFFFF	65535	Activate all debugging options

LOG FILE

The simulation network stores all pertinent command requests in a common log file with a temporary file name. The log file contains simulation network information that is stored there by the simulation manager. The **sn90_close_transaction_log()** function instructs the simulation network to close and rename the log file. A new temporary log file is opened at this time.

Log files are in ASCII format. Log files are available for viewing and printing, and are accessible to other tasks. The format of log file entries is:

mm/dd/yy hh/mm/ss task_ID code description

where:

mm/dd/yy	Month, day and year of the log entry.
hh/mm/ss	Hour, minute and second of the log entry.
task_ID	Name of the task associated with the message.
code	Numeric code assigned to the log message (refer to Table 6-3).
description	Details about the logged condition.

Table 6-3 lists the possible log entry codes and the messages that generate them.

Table 6-3. Log File Entry Codes

Code	Message Type	Code	Message Type	Code	Message Type
1	override add	103	slave malf add error	205	comm error
2	override clear	104	slave malf clear error	206	static snap error
3	slave malf add	105	channel malf add error	207	dynamic snap error
4	slave malf clear	106	channel malf clear error	208	restore done
5	channel malf add	107	run error	209	restore file error
6	channel malf clear	108	freeze error	210	point not found
7	run	109	restore error	211	task not found
8	freeze	110	save error	212	add task
9	restore	111	set sim time error	213	delete task
10	save	112	close log error	214	add point
11	set sim time	113	end simulation error	215	io group
12	close log	201	misc log entry	216	end simulation complete
13	end simulation	202	out of memory	217	bad activity
101	override add error	203	error	218	warning
102	override clear error	204	unexpected msg error		

The sn90_status_code message returns a status code for network and I/O expander bus messages. Table 6-4 provides a listing of these messages.

Table 6-4. Network and I/O Expander Bus Messages

Code	Message	Description
MFP Module Errors		
0000	NO_ERROR	Operation was successful
0001	WRONG_SN_SAVE_TYPE	Unexpected type of snapshot save was requested
0003	WRONG_SN_DATA_TYPE	Unexpected type of snapshot data was requested. This may indicate that a snapshot data file is incompatible with the current MFP firmware revision.
0004	REQ_MODE_CHANGE_INVALID	Invalid MFP mode change request was received. This can occur if an MFP is requested to change to a mode it is already in.
0005	BLOCK_MANAGER_BUSY	MFP's block manager task is busy and cannot perform the requested operation. This can occur when a mode change is requested.
0006	MFP_REQUESTED_TO_STOP	The MFP received a new command after it received a command to halt.
0007	MFP_SHUTTING_DOWN	The MFP received a new command during its shutdown processing.
0008	MFP_IN_CONFIGURE_MODE	The MFP was in configure mode when it received a command that expected it to be in execute mode.
0009	MFP_IN_EXECUTE_MODE	The MFP was in execute mode when it received a command that expected it to be in configure mode.
000A	MFP_IN_ERROR_MODE	The MFP did not respond to a command. This could indicate various conditions, including an error condition within the MFP, the MFP set for normal operation rather than simulation, or the lack of an expander bus dipshunt between the MFP and the USM module.
000B	FILE_SYSTEM_ERROR	A file could not be accessed during a snapshot operation. Typically, this signals an attempt to restore a file into an MFP module in which that file is marked read-only.
000C	MESSAGE_NOT_ALLOWED_IN_MFP_MODE	A message was received that is invalid in the MFP's current mode.
000D	CHECKSUM_ERROR	A checksum indicates that a file has been corrupted.
000E	MESSAGE_NOT_ALLOWED_IN_STARTUP	A message was received that the MFP cannot process during its startup period, which is typically 15 seconds after the RUN command is issued.
000F	MFP_NOT_FROZEN_NO_RESTORE	A snapshot restore command was received while the simulation is running.
0010	MFP_ALREADY_FROZEN	A freeze command was received when the MFP was already in a frozen state.
0011	SLAVE_NOT_SIMULATED	A malfunction or override command was directed to a virtual slave, which cannot accept such commands.
0012	WRONG_DATA_IN_MESSAGE	A malfunction or override message contained an invalid action code.
0013	COMMAND_TIME_OUT	A command was not executed in the time allotted for it.
0014	MESS_NOT_ALL_IN_CONF	The requested operation is not allowed during on-line configuration. This would occur if a snapshot restore was requested while on-line configuration was in progress.
0015	BLOCK_NOT_DEFINED	The block number specified for a virtual slave does not exist in the MFP configuration.

Table 6-4. Network and I/O Expander Bus Messages (Continued)

Code	Message	Description
0016	BLOCK_TYPE_CONFLICT	The block number specified for a virtual slave is digital while the channel in the slave is allocated for analog values, or vice versa.
0017	MFP_NOT_FROZEN_NO_ VIRTUAL_SLAVE	The maximum number of virtual slaves has already been established. No more are allowed.
0018	OUR_ERROR	Internal error. Contact Bailey technical support.
USM Module Errors		
0032	RT_NACK	USM has failed to open a TCP/IP connection.
0033	SEND_VALUES_NOT_ RECEIVED	Input values for the correspondent input group have not been received in the specified time.
0034	XBUS_STALL	A bus stall condition has been detected.
0035	WRONG_SN_DATA_REQ	The requested snapshot data are not available.
0036	PT_ERROR	A point defined in the point table message doesn't match any entry of the slave map.
0037	UNK_LAN_MSG	A TCP/IP unknown message has been received.
0038	SOCK_ERROR	An error has been detected over a socket, bind, or listen operation.
0039	NO_TX	An unpredictable error has been detected on a LAN transmission.
003A	RT_NOTC	A routing tables message has been received while the USM is in on-line not configured mode.
003B	IOG_OV	IO_GROUP entries exceed the maximum allowed number.
003C	INTERNAL_USM_ERROR	Queue error or unknown IJ message
003D	UNK_XBUS_MESS	Unknown expander bus message.
003E	UNKN_BOX	Wrong MAC_STA box.
003F	VIR_SLA_ERR	Virtual slave error.
0040	QUEUE_ERM	Queue full error.
00FF	UNHANDLED_ERROR	USM unknown error.

Sample log file

The following is a copy of a log file generated during a sample run of a simulation. The simulation manager program was started, an application program dedicated to setting up the points within the simulation manager was run, two clients were started, a run command was issued, a snapshot was saved, a point was put into malfunction, the malfunction was cleared, an override value was established for a point and then cleared, the snapshot was restored, and the simulation was stopped.

The line numbers are not a part of the log file as it was generated. They were added for reference.

```
1  01/28/97 13:00:21 sm      201 Opened log file './sm.000'
2  01/28/97 13:00:21 sm      201 SN90 Rev 1.51 November, 1996
3  01/28/97 13:00:21 sm      201 Reading config file sm.cfg
4  01/28/97 13:00:21 sm      201 Set value Client_freezeTimeout=90.000
5  01/28/97 13:00:21 sm      201 Set value USM_LAN_receive_buffer=56
```

```
 6 01/28/97 13:00:21 sm                  201 Set value USM_watchDogCount=10
 7 01/28/97 13:00:21 sm                  201 Set value USM_watchDogInterval=30
 8 01/28/97 13:00:21 sm                  201 Set value StartupTime=120.000
 9 01/28/97 13:00:21 sm                  201 Set value RestoreDelay=0.000
10 01/28/97 13:00:21 sm                  201 Start Simulation
11 01/28/97 13:00:21 sm                  201 debug flags = 2 (2)
12 01/28/97 13:00:21 sm                  201 Resources = 1024
13 01/28/97 13:00:21 ACCEPT              212 New task: port 2714 (A9Ah)
14 01/28/97 13:00:21 sm                  201 keyPort=0 c_keyPort=
15 01/28/97 13:00:21 UDP                 201 Subnet mask=FFFFFFFF
16 01/28/97 13:00:21 sm                  201 Read key on serial port ()
17 01/28/97 13:00:25 sm                  201 2 USM license limit (key ID: 50004)
18 01/28/97 13:00:52 sm                  212 Register Task REMOTECONFIG
19 01/28/97 13:00:52 REMOTECONFIG        212 New task: port 2716 (A9Ch)
20 01/28/97 13:00:52 REMOTECONFIG        201 API v0.30 Apr 30, 1996
21 01/28/97 13:01:27 USM02               201 USM state changed to STARTUP (1,0)
22 01/28/97 13:01:27 USM02               201 IP address 10.33.1.202
23 01/28/97 13:01:27 USM03               201 USM state changed to STARTUP (1,0)
24 01/28/97 13:01:27 USM03               201 IP address 10.33.1.203
25 01/28/97 13:01:39 REMOTECONFIG        213 Task disconnected
26 01/28/97 13:01:39 sm                  201 Cleaning up REMOTECONFIG
27 01/28/97 13:01:43 USM02               201 USM state changed to ONLINE_NOTCONF (1,0)
28 01/28/97 13:01:43 USM02               201 Connect -> sm @ 10.33.1.108:2714
29 01/28/97 13:01:43 USM02               201 Routing table sent init=1 port=1 eobs=1 1
30 01/28/97 13:01:43 sm                  212 Register Task USM02
31 01/28/97 13:01:44 USM02               201 MFP mode changed to Execute (112E8091) (13)
32 01/28/97 13:01:44 USM02               201 Remembering to save static (activity=13)
33 01/28/97 13:01:44 sm                  212 Register Task USM02
34 01/28/97 13:01:44 USM02               201 Routing table ack=1.79s
35 01/28/97 13:01:45 USM02               201 Slave map ack=0.53s
36 01/28/97 13:01:45 USM02               201 Point tables sent count=13
37 01/28/97 13:01:46 USM02               201 Point table ack=0.53s
38 01/28/97 13:01:46 USM02                10 Saving static configuration snapshot data
39 01/28/97 13:01:46 USM02               201 Saving static data
40 01/28/97 13:01:47 USM02               201 Static save ack=1.11s
41 01/28/97 13:01:47 USM03               201 USM state changed to ONLINE_NOTCONF (1,0)
42 01/28/97 13:01:47 USM03               201 Connect -> sm @ 10.33.1.108:2714
43 01/28/97 13:01:47 USM03               201 Routing table sent init=1 port=1 eobs=1 1
44 01/28/97 13:01:48 sm                  212 Register Task USM03
45 01/28/97 13:01:48 USM03               201 MFP mode changed to Execute (1A7D575A) (13)
46 01/28/97 13:01:48 USM03               201 Remembering to save static (activity=13)
47 01/28/97 13:01:48 sm                  212 Register Task USM03
48 01/28/97 13:01:49 USM03               201 Routing table ack=1.73s
49 01/28/97 13:01:50 USM03               201 Slave map ack=0.51s
50 01/28/97 13:01:51 USM03               201 Point tables sent count=337
51 01/28/97 13:01:53 USM03               201 Point table ack=2.55s
52 01/28/97 13:01:54 USM03                10 Saving static configuration snapshot data
53 01/28/97 13:01:54 USM03               201 Saving static data
54 01/28/97 13:01:54 USM03               201 Static save ack=1.07s
55 01/28/97 13:01:54 USM02               201 Static data available
56 01/28/97 13:01:55 USM02               201 Snapshot save complete (0)
57 01/28/97 13:02:07 USM03               201 Static data available
58 01/28/97 13:02:17 USM03               201 Static data available
59 01/28/97 13:02:18 USM03               201 Snapshot save complete (0)
60 01/28/97 13:02:18 sm                  201 USM startup sequence complete
61 01/28/97 13:02:35 USM02               201 USM state changed to ONLINE (0,1)
62 01/28/97 13:02:36 USM02               201 Firmware version B.0d
63 01/28/97 13:02:42 USM03               201 USM state changed to ONLINE (0,1)
64 01/28/97 13:02:42 USM03               201 Firmware version B.0d
65 01/28/97 13:02:52 sm                  218 Simulation startup complete
66 01/28/97 13:02:52 sm                  201 0 clients are ready to run
67 01/28/97 13:02:57 sm                  212 Register Task CLIENT5D
68 01/28/97 13:02:57 CLIENT5D            212 New task: port 2723 (AA3h)
69 01/28/97 13:02:57 CLIENT5D            201 API v0.30 Apr 30, 1996
70 01/28/97 13:02:57 CLIENT5D            201 Connect -> USM03 @ 10.33.1.203:1052 (6)
```

```
 71  01/28/97 13:02:57 CLIENT5D      201 Routing table sent init=1 port=1 eobs=1 0
 72  01/28/97 13:02:58 CLIENT5D      201 Routing table received
 73  01/28/97 13:02:58 CLIENT5D      201 Routing table ack=1.02s
 74  01/28/97 13:02:58 CLIENT5D      215 Group ID='0001' send IO=250ms pts=7
 75  01/28/97 13:02:58 CLIENT5D      215 Group ID='0002' recv IO=250ms pts=3
 76  01/28/97 13:02:59 USM03         201 Connect -> CLIENT5D @ 10.33.1.108:2723 (3)
 77  01/28/97 13:02:59 USM03         201 Routing table sent init=0 port=1 eobs=1 1
 78  01/28/97 13:02:59 USM03         201 Routing table ack=0.51s
 79  01/28/97 13:02:59 sm            201 1 clients are ready to run
 80  01/28/97 13:02:59 USM03         215 Group ID='0001' recv IO=250ms pts=7
 81  01/28/97 13:02:59 USM03         215 Group ID='0002' send IO=250ms pts=3
 82  01/28/97 13:03:02 sm            212 Register Task CLIENT5e
 83  01/28/97 13:03:02 CLIENT5e      212 New task: port 2727 (AA7h)
 84  01/28/97 13:03:03 CLIENT5e      201 API v0.30 Apr 30, 1996
 85  01/28/97 13:03:03 CLIENT5e      201 Connect -> USM02 @ 10.33.1.202:1052 (6)
 86  01/28/97 13:03:03 CLIENT5e      201 Routing table sent init=1 port=1 eobs=1 0
 87  01/28/97 13:03:04 CLIENT5e      201 Routing table received
 88  01/28/97 13:03:04 CLIENT5e      201 Routing table ack=1.02s
 89  01/28/97 13:03:04 CLIENT5e      215 Group ID='0004' send IO=250ms pts=7
 90  01/28/97 13:03:04 CLIENT5e      215 Group ID='0003' recv IO=250ms pts=1
 91  01/28/97 13:03:04 CLIENT5e      215 Group ID='0005' recv IO=250ms pts=3
 92  01/28/97 13:03:04 USM03         201 Connect -> CLIENT5e @ 10.33.1.108:2727 (3)
 93  01/28/97 13:03:04 USM03         201 Routing table sent init=0 port=1 eobs=1 1
 94  01/28/97 13:03:05 USM03         201 Routing table ack=0.51s
 95  01/28/97 13:03:05 USM03         215 Group ID='0003' send IO=250ms pts=1
 96  01/28/97 13:03:05 USM02         201 Connect -> CLIENT5e @ 10.33.1.108:2727 (3)
 97  01/28/97 13:03:05 USM02         201 Routing table sent init=0 port=1 eobs=1 1
 98  01/28/97 13:03:05 USM02         201 Routing table ack=0.51s
 99  01/28/97 13:03:05 USM0          215 Group ID='0004' recv IO=250ms pts=7
100  01/28/97 13:03:05 USM02         215 Group ID='0005' send IO=250ms pts=3
101  01/28/97 13:03:36 sm            212 Register Task console
102  01/28/97 13:03:36 console       212 New task: port 2731 (AABh)
103  01/28/97 13:03:36 console       201 API v0.30 Apr 30, 1996
104  01/28/97 13:03:47 sm            7 1.000000 Run rate requested
105  01/28/97 13:04:04 console       201 sn90_get_message: Long delay 14.65s
106  01/28/97 13:04:52 console       10 sample_snap: Snapshot save requested
107  01/28/97 13:04:53 console       201 sn90_get_message: Long delay 48.49s
108  01/28/97 13:04:56 USM02         201 Snapshot save complete (0)
109  01/28/97 13:07:15 USM03         201 Snapshot save complete (0)
110  01/28/97 13:07:15 sm            10 Snapshot save done
111  01/28/97 13:07:38 console       201 sn90_get_message: Long delay 42.96s
112  01/28/97 13:08:00 USM03         5 Setting malfunction on point AO_1
113  01/28/97 13:08:01 console       201 sn90_get_message: Long delay 22.51s
114  01/28/97 13:08:12 USM03         6 Clearing malfunction on point AO_1
115  01/28/97 13:08:21 console       201 sn90_get_message: Long delay 8.17s
116  01/28/97 13:08:28 console       1 50.000000  0.000000  1  AO_1   Override
117  01/28/97 13:08:29 console       201 sn90_get_message: Long delay 8.33s
118  01/28/97 13:09:07 console       201 sn90_get_message: Long delay 38.29s
119  01/28/97 13:09:09 console       2 0.000000  0.000000  0  AO_1   Override
120  01/28/97 13:10:36 console       201 sn90_get_message: Long delay 86.27s
121  01/28/97 13:10:48 console       9 sample_snap: Full restore of all requested
122  01/28/97 13:10:48 sm            201 Freezing system (1)
123  01/28/97 13:10:49 console       201 sn90_get_message: Long delay 12.29s
124  01/28/97 13:10:50 sm            201 CLIENT5e finished with restore
125  01/28/97 13:10:50 sm            201 CLIENT5D finished with restore
126  01/28/97 13:10:51 sm            201 console finished with restore
127  01/28/97 13:10:53 USM02         201 MFP mode changed to Configure (0) (4)
128  01/28/97 13:11:02 USM03         201 MFP mode changed to Configure (0) (4)
129  01/28/97 13:11:12 USM02         201 MFP mode changed to Execute (112E8091) (4)
130  01/28/97 13:11:15 USM02         201 Snapshot restore complete
131  01/28/97 13:11:15 sm            201 USM02 finished with restore
132  01/28/97 13:12:42 USM03         201 MFP mode changed to Execute (1A7D575A) (4)
133  01/28/97 13:15:01 USM03         201 Snapshot restore complete
134  01/28/97 13:15:01 sm            201 USM03 finished with restore
135  01/28/97 13:15:01 sm            208 Snapshot restore done
```

```
136 01/28/97 13:15:12 console     201 sn90_get_message: Long delay 142.27s
137 01/28/97 13:15:14 sm          7 1.000000 Run rate requested
138 01/28/97 13:15:28 console     201 sn90_get_message: Long delay 10.99s
139 01/28/97 13:15:28 console     8 Freeze requested (1)
140 01/28/97 13:15:28 sm          201 Freezing system (1)
141 01/28/97 13:15:34 console     13 End of simulation requested
142 01/28/97 13:15:34 sm          201 Cleaning up CLIENT5D
143 01/28/97 13:15:35 sm          201 Cleaning up CLIENT5e
144 01/28/97 13:15:56 sm          201 Cleaning up USM02
145 01/28/97 13:16:16 sm          201 Cleaning up USM03
146 01/28/97 13:16:16 sm          201 End Simulation
147 01/28/97 13:16:16 sm          201 Cleaning up console
148 01/28/97 13:16:16 sm          201 System shutdown
149 01/28/97 13:16:17 sm          201 Shutdown complete
150 01/28/97 13:16:17 sm          201 Closed log file './sm.000'
```

Simulation manager startup (lines 1-17)

The simulation manager program is started. The log file's name is given, and the version of the simulation manager program is listed. If an ***sm.cfg*** file is found, it is read, and any values used from it are copied into the log. The debug value is reported. The maximum permissible number of USM modules that the simulation manager can connect to is reported.

Simulation manager configuration (lines 18-66)

An application program named REMOTECONFIG is started. REMOTECONFIG's purpose is to establish the USMs, I/O modules, and points that will be available in the simulation. Note that REMOTECONFIG is not necessarily the name of the configuration program's executable file. Instead, it is the name that was specified in the **sn90_open()** function call. The REMOTECONFIG program issues a collection of **sn90_establish_USM_*XXXX*()** type function calls. After all of them are issued, **sn90_establish_USM_done()** is called and the program exits.

In lines 18-20, the REMOTECONFIG program is registered with the simulation manager and reports the version of the application programming interface that it is using.

The REMOTECONFIG program establishes two USM modules, that will be named USM02 and USM03 (due to their IP addresses). Tasks for the two modules are started and the IP addresses that the simulation manager will expect them to have are reported.

Lines 25 and 26 report that the simulation manager has recognized that REMOTECONFIG has halted, and connections to it are shut down.

In lines 27-40, USM02's startup activities are reported. A connection is established between the simulation manager and the USM module, and its port number is reported in line 28. Each USM module will have its own task that is registered with the simulation manager, as application programs do. USM02's registration is noted in line 30.

A routing table that contains port numbers of known tasks is transmitted to the USM module and acknowledged. It is fol-

lowed by a map of the I/O modules that are to be simulated in that USM module, which is acknowledged. After that, the list of the points in those I/O module is sent. The total number of points is reported, as is the receipt of an acknowledgment of the list.

In lines 38-40, the start of the process of reading static snapshot data from USM02's MFP module is reported.

While USM02's MFP module's static snapshot data is being received, USM03 begins going through the same process, as shown in lines 41-54.

In line 55, USM02 reports that it has the static snapshot data from its MFP module, and is ready to send it. In line 56, the completion of the snapshot data reception is reported. In lines 57 and 58, the completion of USM03's static snapshot save process is reported.

After USM03's snapshot save is finished, there is no more USM module initialization work to be performed, and the *USM startup sequence complete* message appears in the log file. It is generally recommended that this is the time to start any process modeling or other programs that are going to be running in conjunction with the simulation. In this case, the application programs were not started until slightly later.

Line 65 shows the *Simulation startup complete* message. This message appears after the startup time expires, if all USM modules have finished initializing. At the same time as this message appears, a message reporting the number of application programs that will do point input/output is reported.

Client startup (lines 67-104)

Lines 67 through 79 report the startup of the first application program, whose name is CLIENT5D. The program is registered with the simulation manager, and routing tables are updated to include it. Points that the client will use are established using **sn90_establish_point()**. After the call to **sn90_establish_point_done()**, the program's points are sorted by destination, direction and update frequency, and I/O groups are created and listed. Each I/O group has two tasks associated with it, one to send it and one to receive it. Both tasks report the creation of the I/O groups. CLIENT5D only talks with USM03, and has one I/O group going in each direction.

Lines 80-100 report the startup of the second application program, CLIENT5E. This program communicates with both USM modules. Routing table and I/O group messages describing the connections are generated accordingly.

Lines 101-103 report the startup of a client named ***console***. This is an instance of the example program ***uc.c***. This program does not do point input/output, so there are no routing table

or I/O group messages. Its purpose is to control the simulation. Its first action is to issue a run command as reported in line 104. A few seconds after that message appears, the application programs and MFP modules begin exchanging data via the USM modules.

Snapshot save (lines 105-110)

The console program was used to issue a command to save all snapshot data. The command is acknowledged. After snapshot saving is complete for each USM module, a message is written to the log file. After the last task acknowledgment is received, the *Snapshot save done* message is issued.

The *Long delay* message that appears frequently in this log file is due to the fact that the ***uc*** program does not check for messages at frequent intervals. When an application program calls **sn90_get_messages()**, an internal timer is checked to find out how long it has been since that function was last called. If the delay is longer than six seconds, a *Long delay* message appears. This is a warning that a process may not have been keeping up with commands that have been sent to it. The six second interval is not configurable.

Malfunction setting and clearing (lines 111-114)

The console task was used to set and then clear a malfunction on a point named AO_1.

Override setting and clearing (lines 115-119)

The console task was used to set and then clear an override value on point AO_1. The override set message is characterized by a message code of 1, the override value, the actual value of the point, the quality check flag that was specified in the **sn90_override_add()** function call, and the name of the overridden point. The override clear message is characterized by a message code of 2. All other values in the log message are always 0.

Snapshot restore (lines 120-137)

The console task was used to request the restoration of the snapshot that was previously saved. A message reporting the request is logged. The simulation is automatically frozen and a report of the freeze is also logged. Acknowledgments from the application tasks are reported as they are received (they are received quickly since these tasks do not do anything with a snapshot restore command except to acknowledge it). The modules are changed to configure mode to accept static data, including block specifications and data files, and then they are put back into execute mode. After that, dynamic data is sent to them. The USM tasks report completion of snapshot restoration. After all tasks (both application and USM module) have

acknowledged completion, the *Snapshot restore done* message is logged.

> **NOTE:** After a full restore, the simulation is left frozen. In the sample log file, the run command reported on line 137 was a response to a command from the CONSOLE task. If a restore is requested for a single task, the run command will be automatically issued after the restore is completed. This difference may be removed in future revisions of the simulation LAN and API system.

Simulation shutdown (lines 138-150)

Before shutting down the simulation, the simulation was frozen by a command sent from the console. This was done to avoid *NO TX* errors that appear in the log file if an end simulation command is received while the simulation is running. This is due to the fact that application programs are shut down before the simulation manager and there is some time when the simulation manager is expecting data from the application programs but the connections to the application programs have been broken.

The end simulation command is by the simulation manager on line 141. The simulation manager passes it on to the application programs, reporting that each program is being shut down. When all acknowledgments have been received, the simulation manager reports that it is shutting itself down and finally it exits.

COMPILING AND LINKING C APPLICATIONS

Applications written using C language must be compiled using an HP C compiler (not supplied) and linked using an HP C++ compiler (not supplied) or the C++ linker provided.

> **NOTE:** Insure that the main program of the application calls _main() before doing anything else. Also verify the ***sn90_api.h*** file is included in the program.

To compile a C application, type:

cc -c -Ae -I/sn90/include *app_name*.**c** ENTER

where:

app_name Name of the C application file.

To link a compiled C application, type:

CC +a1 +A -o *exec_name app_name*.**o -L/sn90/lib -lsn90** ENTER

where:

exec_name	Name of the executable file created by the link operation.
app_name	Name of the object code file created by the compile operation.

COMPILING AND LINKING C++ APPLICATIONS

Applications written using C++ language must be compiled and linked using the HP C++ compiler (not supplied).

NOTE: Insure that the ***sn90_api.h*** file is included by the program.

To compile and link a C++ application, type:

CC +a1 +A -o *exec_name app_name*.**C -I/sn90/include -L/sn90/lib -lsn90** ENTER

where:

exec_name	Name of the executable file created by the link operation.
app_name	Name of the C application file.

SERVICES

The installation procedure updates the service port listing in /etc/services. The setup script updates the network services file /etc/services. The following entries should be found in /etc/services:

```
# Simulation ports

sn90tm          1050/udp
sn90tm          1051/tcp
sn90chm         1050/udp
sn90chm         1051/tcp
sn90lanrx       1052/tcp
sn90lanrx       1053/udp
sn90USMrx       1052/tcp
sn90USMrx       1053/udp
```

The file provides information of port numbers for specific protocols and service names.

SECTION 7 - ERROR MESSAGES AND RECOVERY

INTRODUCTION

All API functions are capable of returning an error message of type short int (short integer). Table 7-1 lists the possible error messages.

Table 7-1. Error Messages

Error Message	Condition	Corrective Action
SN90_BAD_ARGUMENT	An argument is out of range.	Verify all arguments are properly specified.
SN90_CONNECT_ERROR	An error occurred when simulation manager tried connecting to a task.	Verify the task is running.
SN90_EOF	End of list.	None.
SN90_ERROR	Undefined error.	Contact Bailey Controls Company technical support department.
SN90_INV_REQ_CODE	A receiving task gets an invalid message code.	
SN90_INVALID_SIMULID	Simulation ID not found.	Verify simulation ID is correct.
SN90_INVALID_TASKID	Task ID not found.	Verify task ID is correct.
SN90_MEMORY_FAIL	Memory allocation error.	Verify sufficient memory exists.
SN90_NORMAL	Normal operation.	None.
SN90_NOT_CONNECTED	Connection between tasks broken.	Verify network connections.
SN90_RECEIVE_MSG_ERROR	Corrupted message received.	Contact Bailey Controls Company technical support department.
SN90_SEND_MSG_ERROR	Requesting task unable to send message due to lost connection.	Verify the simulation manager is running.
SN90_TIMEOUT	Simulation manager is not responding.	

SECTION 8 - MESSAGE HANDLING

INTRODUCTION

This section describes the three types of notification messages and provides a listing of them. Also described are the proper responses to these messages.

NOTIFICATION MESSAGES

Messages received by the **sn90_get_message()** function that contain parameters that are of no interest to the task can be responded to by the task using the **sn90_ack_message()** function. For example, the task receives an SN90_SNAPSHOT_SAVE notification message but the task does not perform any snapshot activity. Instead of requesting the snapshot ID using the **sn90_get_snapshot_save()** function, the task issues the **sn90_ack_message()** function. The simulation network determines that the SN90_SNAPSHOT_SAVE_ACK message is the appropriate reply and sends it to the task. The SN90_SNAPSHOT_SAVE notification message is discarded without unpacking the remaining contents.

Alternatively, after receiving the SN90_SNAPSHOT_SAVE notification message, the task could issue the **sn90_save_snapshot_ack()** function that causes the simulation network to discard the SN90_SNAPSHOT_SAVE notification message and to send the SN90_SNAPSHOT_SAVE_ACK reply to the task.

Notification messages are divided into three categories: command notifications, acknowledgment notifications, and collection notifications.

Command Notifications

Command notification messages and the appropriate responses are described in Table 8-1.

Table 8-1. Command Notification Messages

Message	Response
SN90_END_OF_SIMULATION	Issue the **sn90_ack_message()** function.
SN90_END_OF_SIMULATION_DONE	If the message status is SN90_ERROR, check the log file to determine which task had a failure.
SN90_FREEZE	Issue the **sn90_freeze_ack()** function.
SN90_FREEZE_DONE	If the message status is SN90_ERROR, use the **sn90_collect_task_status()** and **sn90_next_task_status()** functions to determine which task failed, and nature of the problem.

Table 8-1. Command Notification Messages (Continued)

Message	Response
SN90_RUN	Issue the **sn90_get_run()** function to receive the execution rate. After taking any appropriate run action, issue the **sn90_run_ack()** function.
	If no special actions are required, issue the **sn90_run_ack()** function.
SN90_RUN_DONE	If the message status is SN90_ERROR, issue the **sn90_collect_task_status()** function to determine which task had a failure.
SN90_SIMULATION_TIME	Issue the **sn90_get_simulation_time()** function to receive the simulation time parameters. After performing any required actions, issue the **sn90_simulation_time_ack()** function.
	If no special actions are required, issue the **sn90_simulation_time_ack()** function.
SN90_SNAPSHOT_RESTORE	Issue the **sn90_get_snapshot_restore()** function to receive the snapshot ID. Perform any appropriate snapshot restore actions and then issue the **sn90_snapshot_restore_ack()** function.
	If no special actions are required, issue the **sn90_snapshot_restore_ack()** function.
SN90_SNAPSHOT_RESTORE_DONE	If the message status is SN90_ERROR, use the **sn90_collect_task_status()** and **sn90_next_task_status()** functions to determine which task failed, and what the problem was.
SN90_SNAPSHOT_SAVE	Issue the **sn90_get_snapshot_save()** function to receive the snapshot ID. Perform any appropriate snapshot save actions and then issue the **sn90_snapshot_save_ack()** function.
	If no special actions are required, issue the **sn90_snapshot_save_ack()** function.
SN90_SNAPSHOT_SAVE_DONE	If the message status is SN90_ERROR, use the **sn90_collect_task_status()** and **sn90_next_task_status()** functions to determine which task failed, and what the problem was.

Acknowledgment Notifications

Acknowledgment notification messages do not require any response to be sent by the application task to the simulation manager. Acknowledgment messages are listed in Table 8-2.

Table 8-2. Acknowledgment Notification Messages

Message	Response
SN90_FREEZE_ACK	None
SN90_MALFUNCTION_ACK	
SN90_OVERRIDE_ACK	
SN90_RUN_ACK	
SN90_SNAPSHOT_RESTORE_ACK	
SN90_SNAPSHOT_SAVE_ACK	

Collection Notifications

Collection notification messages and the appropriate responses are described in Table 8-3.

Table 8-3. Collection Notification Messages

Message	Response
SN90_CHANNEL_MALFUNCTIONS_ COLLECTED	Issue the **sn90_next_channel_malf()** function to receive the parameters of the first or next channel malfunction collected. Repeat this function until the SN90_EOF message is returned. **NOTE:** Issuing the **sn90_collect_channel_malf()** function before the SN90_EOF message is received will erase any remaining channel malfunctions on the queue.
SN90_IOMOD_MALFUNCTIONS_ COLLECTED	Issue the **sn90_next_iomodule_malf()** function to receive the parameters of the first or next I/O module malfunction collected. Repeat this function until the SN90_EOF message is returned. **NOTE:** Issuing the **sn90_collect_iomodule_malf()** function before the SN90_EOF message is received will erase any remaining I/O module malfunctions on the queue.
SN90_OVERRIDES_COLLECTED	Issue the **sn90_next_override()** function to receive the parameters of the first or next active override collected. Repeat this function until the SN90_EOF message is returned. **NOTE:** Issuing the **sn90_collect_overrides()** function before the SN90_EOF message is received will erase any remaining overrides on the queue.
SN90_POINT_NAMES_COLLECTED	Issue the **sn90_next_pointname()** function to receive the parameters of the first or next point name collected. Repeat this function until the SN90_EOF message is returned. **NOTE:** Issuing the **sn90_collect_pointnames()** function before the SN90_EOF message is received will erase any remaining point names on the queue.
SN90_SNAPSHOT_COLLECTED	Issue the **sn90_get_snapshot_time()** function to receive the simulation time and local time when the snapshot was collected. Issue the **sn90_next_snapshot_task()** function to receive the task ID and task status for that task. Repeat this function until the SN90_EOF message is returned. **NOTE:** Issuing the **sn90_collect_snapshot()** function before the SN90_EOF message is received will erase any remaining items on the queue.
SN90_TASK_STATUS_COLLECTED	Issue the **sn90_next_task_status()** function to receive the parameters of the first or next task status collected. Repeat this function until the SN90_EOF message is returned. **NOTE:** Issuing the **sn90_collect_task_status()** function before the SN90_EOF message is received will erase any remaining task status on queue.

APPENDIX A - QUICK REFERENCE

INTRODUCTION

This section provides a quick reference guide to the API functions. Table A-1 lists the modules and function codes that support I/O module malfunctions and the types of I/O module malfunctions available. Table A-2 lists the available channel addresses for each I/O module type.

Table A-1. Possible I/O Module Malfunctions

I/O Module	Function Codes Required	Malfunction Code					Defaults Supplied By
		1	2	3	4	5	
IMASI02, IMFBS01	132	X	X	X[1]			N/A
IMASI03	215, 216, or 217	X	X	X[6]			N/A
IMASO01	149[2]	X	X	X[1,5]			MFP module
IMCIS02, IMCIS12, IMQRS02	79[2]	X	X	X[1,5]			Dipswitches
IMDSI02, IMDSI12/13/14/15	84 or 114	X[3,7]					N/A
IMDSM04	102, 103, 104, or 109	X		X[1]	X		N/A
IMDSM05	83, 84, 114, or 115	X[3,7]					Dipswitches
IMDSO01/02/03, IMDSO15	83 or 115	X[3,7]				X	MFP module
IMDSO04, IMDSO14	83 or 115	X[3,7]				X	MFP module
IMFCS01	145	X		X[1]			N/A
IMHSS01	150[2]	X[7]					Intrinsic[4]
IMRIO02	146	X[8]					N/A
IMSET01, IMSED01	242	X		X[1,9]			N/A

NOTES:

1. Channel malfunction generated in response to sn90_channel_malf_add() function call.
2. This function code reads feedback values during start-up and override.
3. Grouped channels: all 8 channels in the affected group will show bad quality after a call to sn90_channel_malf_add() for any channel in the group.
4. If this module loses its control module, it enters emergency manual mode (hold).
5. This malfunction will be reported if an analog output point from this module is overridden.
6. A call to sn90_channel_malf_add() for one of this module's channels will result in this malfunction being reported for the affected FC 216 block. A call to sn90_iomodule_malf_add() for this module will result in this malfunction being reported for the modules FC 215 block.
7. This malfunction code will be reported if sn90_channel_malf_add() is called for any channels in this I/O module type.
8. A malfunction can only be simulated for an IMRIO02 configured as a Remote Master Processor. All channels of all I/O modules communicating through the simulated RMP module will report a no response/wrong type error when the RMP is malfunctioning.
9. Grouped channels: all 16 channels in this I/O module will show zero and bad quality after a call to sn90_channel_malf_add() for any channel in the I/O module.

Table A-2. I/O Module Channel Addresses

I/O Module Type	Channel Type	Available Channel Addresses
SN90_ASI02	Analog input	0 - 14
SN90_ASI03	Analog input	0 - 15
SN90_ASO01	Analog output	0 - 13
SN90_CIS02	Analog input Analog output Digital input Digital output	0 - 3 4 - 5 6 - 8 9 - 12
SN90_DSI01	Digital input	0 - 15
SN90_DSI02	Digital input	0 - 15
SN90_DSM03	Digital input	0 - 15
SN90_DSM04	Analog input	0 - 7
SN90_DSM05_I	Digital input	0 - 15
SN90_DSM05_IO	Digital input Digital output	0 - 7 8 - 15
SN90_DSM05_O	Digital output	0 - 15
SN90_DSM05_OI	Digital output Digital input	0 - 7 8 - 15
SN90_DSO01	Digital output	0 - 7
SN90_DSO02	Digital output	0 - 7
SN90_DSO03	Digital output	0 - 7
SN90_DSO04	Digital output	0 - 15
SN90_FBS01	Analog input	0-14
SN90_FCS01	Analog input	0
SN90_HSS01	Analog output	0
SN90_QRS02	Analog input Analog output Digital input Digital output	0 - 3 4 - 5 6 - 8 9 - 12
SN90_RIO02	Remote I/O	None
SN90_SOE	Digital Input	0-15
SN90_VIRT_IO	Analog output Digital output	0 - 7 8 - 15

APPENDIX B - EXAMPLE APPLICATIONS

INTRODUCTION

Example application programs provided with the simulation network API software are described in Table B-1.

Table B-1. Example Applications

Application	Description
USMconfig	A USM configuration program that reads USM, I/O module and point information from a file and configures a simulation accordingly.
client1	A sample client program that exchanges data with client6.
client6	A sample client program that exchanges data with client1.
diag	A program to extract diagnostic data from a USM module. **NOTE:** The diag program utilizes an unsupported feature of the Simulation Network API library. This feature may be modified or deleted without notice.
end	A program that performs an orderly shutdown of the simulation system.
estusm	A sample program that establishes a USM module, one I/O module, and its points.
io	A sample client program.
log	A client program that displays messages that are being written to the simulation manager's log file.
shutdown	A program that performs an emergency shutdown of the simulation system. This program ensures that USM modules are cleanly shut down in the event of an upset to the system.
taskio	A sample client program that uses the points established by estusm.
taskstatus	A client program that reports the status of all simulation tasks currently running.
uc	A utility console program. This program provides a simple user interface for controlling and monitoring a simulation. **NOTE:** The utility console program violates the requirement that client programs call **sn90_get_messages()** regularly. If another client program issues a command that requires other tasks to acknowledge it, such as a run command, the utility console program will not respond in time, and time-out errors will occur.

The source code for the taskio, estusm and taskstatus are presented here as examples.

PROGRAM ONE - CLIENT_TASKIO

This program establishes points and performs I/O operations with a USM module and is stored in the ***taskio.C*** file.

This program issues the **sn90_open()** function to connect the task manager task with the name CLIENT_TASKIO and then issues the **sn90_register()** function to connect with the simulation provided as a runtime argument. Ten points are established for communication to the USM module. The USM

module must also have these same point names established. For this program, a task such as program two (file ***estusm.C***) is used to establish the USM module. If the SN90_END_OF_SIMULATION message is received, the program exits. The following text is a listing of the program code.

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

#include "sn90_api.h"                    /* simulation API header file */

#define PT_COUNT        10               /* maximum number of points for test */
#define TRUE            1
#define FALSE           0

int   ctr = 0;
unsigned int RUN = FALSE;                /*simulation run status */

int   ProcessMessage (short msg_code, short stat);

void main(int argc, char *argv[])
{
      short     msg_code = 0;
      short     msg_status = SN90_NORMAL;
      short     status = 0;
      short     toggle = 1;

      short     ai_1_stat;               /* point status AI_1 */
      short     ai_2_stat;               /* pont status AI_3 */
      short     ai_3_stat;               /* point status AI_2 */
      short     ai_4_stat;               /* point status AI_4 */
      short     di_1_stat;               /* point status DI_1 */
      short     di_2_stat;               /* point status DI_2 */
      short     di_3_stat;               /* point status DI_3 */
      short     ao_1_stat;               /* point status AO_1 */
      short     ao_2_stat;               /* point status AO_2 */
      short     do_1_stat;               /* point status D0_1 */

      float     ai_1;                    /* point value AI_1 */
      float     ai_2;                    /* point value AI_2 */
      float     ai_3;                    /* point value AI_3 */
      float     ao_1;                    /* point value AO_1 */
      float     ao_2;                    /* point value AO_2 */
      float     ai_4;                    /* point value AI_4 */
      char      di_1;                    /* point value DI_1 */
      char      di_2;                    /* point value DI_2 */
      char      di_3;                    /* point value DI_3 */
      char      do_1;                    /* point value DO_1 */

      _main();                           /* required if HP C compiler is used */

      if (argc !=2)
      {
            printf("Usage: CLIENT_TASKIO simulation_name\n");
            return;
      }

      /* this task will be known as CLIENT_TASKIO to the simulation network */

      if ((status = sn90_open("CLIENT_TASKIO")) == SN90_NORMAL)
      {
```

```
status = sn90_register(argv[1]);
if (status != SN90_NORMAL)
      printf("CLIENT_TASKIO: Failed Registering with Simulation Manager\n");
else
{
      printf("Establishing 10 Points \n");

      sn90_establish_point("AI_1", 0, SN90_SEND, SN90_FLOAT, &ai_1, 0, &ai_1_stat);
      sn90_establish_point("AI_2", 0, SN90_SEND, SN90_FLOAT, &ai_2, 0, &ai_2_stat);
      sn90_establish_point("AI_3", 0, SN90_SEND, SN90_FLOAT, &ai_3, 0, &ai_3_stat);
      sn90_establish_point("AI_4", 0, SN90_SEND, SN90_FLOAT, &ai_4, 0, &ai_4_stat);

      sn90_establish_point("AO_1", 250, SN90_RECV, SN90_FLOAT, &ao_1, 0, &ao_1_stat);
      sn90_establish_point("AO_2", 250, SN90_RECV, SN90_FLOAT, &ao_2, 0, &ao_2_stat);
      sn90_establish_point("DO_1", 250, SN90_RECV, SN90_BYTE, &do_1, 0, &do_1_stat);

      sn90_establish_point("DI_1", 0, SN90_SEND, SN90_BYTE, &di_1, 0, &di_1_stat);
      sn90_establish_point("DI_2", 0, SN90_SEND, SN90_BYTE, &di_2, 0, &di_2_stat);
      sn90_establish_point("DI_3", 0, SN90_SEND, SN90_BYTE, &di_3, 0, &di_3_stat);

      /*
      ** This example supplies the address of the point value and the
      ** address of the point status. This method eliminates the need
      ** to call sn90_read_value() and sn90_write_value(). The call
      ** to sn90_update_values() takes care of mapping data from the
      ** network I/O buffer to the appropriate application memory
      ** location. The points here were all established with an
      ** I/O interval of 250 milliseconds.
      */

      if ((status = sn90_establish_point_done()) == SN90_NORMAL)
      {
            printf("Establish Points Successful. Status = %d\n", status);

            ai_2 = 1.0;
            ai_3 = 1.0;
            di_2 = 1;
            di_3 = 1;

            while(1)          /* activity loop */
            {
                  if (RUN)
                  sn90_update_values();          /* Do I/O over the simulation network */

                  ai_1 = ao_1;                   /* Do application activities */
                  ai_4 = ao_2;
                  di_1 = do_1;

                  if (ctr >= 10)   /* print every 10 cycles */
                  {
                        printf("\n%s%s\n", "-------------------------------------", "----------");
                        printf("CLIENT_TASKIO: Input - AO_1 %f, I/O status: %d\n", ao_1, ao_1_stat);
                        printf("CLIENT_TASKIO: Input - AO_2 %f, I/O status: %d\n", ao_2, ao_2_stat);
                        printf("CLIENT_TASKIO: Input - DO_1 %d, I/O status: %d\n\n", do_1,
                            do_1_stat);
                        printf("\n");
                        ctr = 0;

                        if (toggle < 0)
                        {
                            di_2 = 0;
                            di_3 = 0;
                        }
                        else
                        {
```

```
                        di_2 = 1
                        di_3 = 1;
                    }
                    if (ai_2 > 150) ai_2 = 0;
                    if (ai_3 > 150) ai_3 = 0;
                    ai_2 = ai_2 + 1.0;
                    ai_3 = ai_3 + 1.0;
                    toggle = toggle * (-1);

                }
                else
                {
                    ctr++;
                }
                  /* get notification messages */
                status = sn90_get_message(&msg_ccode, &msg_status);

                if (status = SN90_NORMAL)
                {
                    if(ProcessMessage(msg_code, status))
                    {
                            printf("CLIENT_TASKIO: END OF SIMULATION\n");
                            sn90_close(1.0);
                            return;
                    }
                }
                else /* report and take action on error */
                {
                    printf("CLIENT_TASKIO: ERROR, msg_code-->%d, msg_status-->, status=%d\n",
                            msg_code, msg_status, status);


                sn90_delay(.100);       /** delay 100 milliseconds so as not to consume all cpu */

            }   /* end of while loop */
        }
        else
        {
            printf("CLIENT_TASKIO: Establish Points Failed. Status = %d\n", status);
        }
    }
}
else
{
    printf("CLIENT_TASKIO: Failed Registering with Task Manager, Status = %d\n", status);
}
sn90_close(1.0);   /* close with 1 second delay */
return;
}

/*****************************************************************************
**
**   Function:     ProcessMessage
**
**   A routine of this type is provided by the application and it
**   determines how to respond to simulation network notification
**   messages.
**
*****************************************************************************/
int ProcessMessage(short msg_code, short stat)
{
    short status = 0

    switch (msg_code)
    {
```

```
            case SN90_END_OF_SIMULATION:
                sn90_ack_message();
                status = 1;
                return 1;

            case SN90_FREEZE:
                RUN = FALSE;
                sn90_freeze_ack(SN90_NORMAL);
                break;

            case SN90_RUN:
                RUN = TRUE;
                sn90_run_ack(SN90_NORMAL);
                break;

            case SN90_SNAPSHOT_RESTORE:
                {
                    char snapID[SN90_SNAPIDLENGTH+1];
                    sn90_get_snapshot_restore(snapID);
                    printf("Snapshot Restore ID= >%s<\n", snapID);
                    sn90_snapshot_restore_ack(SN90_NORMAL);
                }
                break;

            case SN90_SNAPSHOT_SAVE:
                {
                    char snapID[SN90_SNAPIDLENGTH+1];
                    sn90_get_snapshot_save(snapID);
                    printf("Snapshot Save ID= >%s<\n", snapID);
                    sn90_snapshot_save_ack(SN90_NORMAL);
                }
                break;

            case SN90_NO_MESSAGE:
                break;

            default:
                printf("CLIENT_TASKIO: Received MsgCode->%d, Status->%d\n", msg_code, stat);
                sn90_ack_message();
                break;
        }

        return(status)
}
```

PROGRAM TWO - ESTUSM

This program establishes a USM module and is stored in the ***estusm.C*** file.

This program issues the **sn90_open()** function to connect the task manager task with the name ESTUSM and then issues the **sn90_register()** function to connect with the simulation provided as a runtime argument. The USM module is assigned a TCP/IP address of 255.255.255.255 and task ID of USM01. The USM module has one IMCIS01 I/O module. Points are established for 13 channels. If the SN90_END_OF_ SIMULATION message is received, the program exits. The following text is a listing of the program code.

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

#include "sn90_api.h"                   /* simulation API header file */

int ProcessMessage(short msg_code, short stat);

void main(int argc, char *argv[])
{
      short     msg_code = 0;
      short     msg_status = SN90_NORMAL;
      short     status = 0;

      _main();                          /* required if HP C compiler is used */

      if (argc != 2)
      {
            printf("Usage: ESTSUM simulation_name\n");
            return;
      }

      if ((status = sn90_open("ESTUSM")) == SN90_NORMAL)
      {
            status = sn90_register(argv[1]);

            if (status != SN90_NORMAL)
                  printf("ESTUSM: Failed Registering with SM\n");
            else
            {
                  /* Establish USM */
                  printf("Establishing USM01\n");
                  sn90_establish_USM("255.255.255.255", "USM01");

                  /* Establish USM Slave Xbus address = 2, presence = 1 = simulated, slave type = 13 = CIS,
                        virtual update time = 0 */
                  sn90_establish_USM_iomodule("USM01", 2, 1, 13, 0);

                  /*
                  **  Establish USM points with default status = fail high
                  **
                  **  short  sn90_establish_USM_point(taskID, Xbus Addr,
                  **                        channel, name, default status, pt type,
                  **                        I/O direction, I/O interval, virtual
                  **                        block)
                  **  These parameters could be read in from some data source
                  **  instead of being hard coded as for this example.
                  */

                  sn90_establish_USM_point("USM01", 2, 0,  "AI_1",  1, 4, 0, 500, 0);
                  sn90_establish_USM_point("USM01", 2, 1,  "AI_2",  1, 4, 0, 500, 0);
                  sn90_establish_USM_point("USM01", 2, 2,  "AI_3",  1, 4, 0, 500, 0);
                  sn90_establish_USM_point("USM01", 2, 3,  "AI_4",  1, 4, 0, 500, 0);
                  sn90_establish_USM_point("USM01", 2, 4,  "AO_1",  1, 4, 1,   0, 0);
                  sn90_establish_USM_point("USM01", 2, 5,  "AO_2",  1, 4, 1,   0, 0);
                  sn90_establish_USM_point("USM01", 2, 6,  "DI_1",  1, 1, 0, 500, 0);
                  sn90_establish_USM_point("USM01", 2, 7,  "DI_2",  1, 1, 0, 500, 0);
                  sn90_establish_USM_point("USM01", 2, 8,  "DI_3",  1, 1, 0, 500, 0);
                  sn90_establish_USM_point("USM01", 2, 9,  "DO_1",  1, 1, 1,   0, 0);
                  sn90_establish_USM_point("USM01", 2, 10, "DO_2",  1, 1, 1,   0, 0);
                  sn90_establish_USM_point("USM01", 2, 11, "DO_3",  1, 1, 1,   0, 0);
                  sn90_establish_USM_point("USM01", 2, 12, "DO_4",  1, 1, 1,   0, 0);

                  /* Establish USM Done */
                  status = sn90_establish_USM_done();
```

```
                printf("Establish USM Status = %d\n", status);

                while(1)        /* get message loop */
                {
                    status = sn90_get_message(&msg_code, &msg_status);
                    if (status == SN90_NORMAL)
                    {
                        if (ProcessMessage(msg_code, status))
                        {
                            printf("estusm: END OF SIMULATION\n");
                            sn90_close(1.0);
                            return;
                        }
                    }

                    else        /* report and take action on error */
                    {
                    printf("estusm: sn90_get_message returned ERROR-->%d\n", status);
                    }

                sn90_delay(0.5);        /* delay for 500 milliseconds */
                }
            }
        }
        else
        {
        printf("estusm: Failed Registering with Task Manager, Status = %d\n", status);
        }

        sn90_close(1.0);
        return;
}


/*************************************************************************
**
**  Function:  ProcessMessage
**
**  A routine of this type is provided by the application and it
**  determines how to repond to simulation network notification
**  messages.
**
*************************************************************************/
int ProcessMessage(short msg_code, short stat)
{
        short status = 0;

        switch(msg_code)
        {
            case SN90_RUN:
                sn90_run_ack(SN90_NORMAL);
                printf("estusm: Sending Run Ack.\n");
                break;

            case SN90_SNAPSHOT_SAVE_ACK:
                sn90_snapshot_save_ack(SN90_NORMAL);
                printf("estusm: Sending Snapshot Save Ack.\n");
                break;

            case SN90_END_OF_SIMULATION:
                sn90_ack_message();
                printf("estusm: Sending End Simulation Ack\n");
                status = 1;
                break;
```

```
            case SN90_NO_MESSAGE:
                  break;

            default:
                  if (status = sn90_ack_message() )
                  printf("estusm: sn90_ack_message() status %d\n", status);
                  break;
      }

      return(status);
}
```

PROGRAM THREE - TASKSTATUS

This program collects the status of all tasks in a simulation. Its source code file is ***taskstat.C***. The program illustrates the process of asking that data be collected, and then repeatedly asking for that data until no more is available. Within the main function, the sn90_collect_task_status() function is called. When an SN90_TASK_STATUS_COLLECTED message is received, the sn90_next_task_status() function is called within a loop to gather the requested data. As long as the return value of sn90_next_task_status() is SN90_NORMAL, there is unprocessed information waiting, and the logic remains in the loop. When the function returns SN90_EOF, all available data has been processed and the logic exits the loop.

```
#include <stdio.h>
#include <unistd.h>
#include "sn90_api.h"        /* simulation API header file */

int ProcessMessage(short msg_code, short stat);

void main(int argc, char *argv[])
{
      short      msg_code = 0;
      short      msg_status = SN90_NORMAL;
      short       status =   0;
      char        task_id[SN90_TASKIDLENGTH];

      _main();                  /* required if HP C compiler is used */

      if (argc != 2)
      {
            printf("Usage: taskstatus simulation_name\n");
            return;
      }

      if ((status = sn90_open("TASKSTATUS")) == SN90_NORMAL)
      {
            status = sn90_register(argv[1]);
            if (status != SN90_NORMAL)
                  printf("taskstatus: Failed Registering with SM\n");
            else
            {
                  status = sn90_collect_task_status();

                  if (status != SN90_NORMAL)
```

```
                {
                    printf("Error: sn90_collect_task_status(), status = %d\n", status);
                    return;             /* exit task */
                }
                else
                {
                    while(1)            /* get message loop */
                    {
                        sleep(1);
                        status = sn90_get_message(&msg_code, &msg_status);
                        if (status == SN90_NORMAL)
                            if (ProcessMessage(msg_code, status))
                                return;             /* exit task */
                            else
                                printf("Error: sn90_get_message() status = %d\n", status);
                    }
                }
            }
        }
        else
            printf("taskstatus: Failed Registering with TM, Status = %d\n", status);

        return;
}



/*************************************************************************
**
** Function:  ProcessMessage
**
** A routine of this type is provided by the application and it
** determines how to repond to simulation network notification
** messages.
**
*************************************************************************/

int ProcessMessage(short msg_code, short stat)
    {
        int     returnStatus = SN90_NORMAL;
        short   status = SN90_NORMAL;

        switch(msg_code)
        {
            case SN90_TASK_STATUS_COLLECTED:
                {
                    short       run_status;
                    short       MFP_status;
                    short       comm_status;
                    short       restore_status;
                    short       save_status;
                    char        msg_task_id[SN90_TASKIDLENGTH];

                    while (status == SN90_NORMAL)
                    {
                        status = sn90_next_task_status(msg_task_id, &run_status, &MFP_status,
                                    &comm_status, &restore_status, &save_status);
                        if (status != SN90_EOF)
                            printf("%s%3d, %s%3d, %s%4d, %s%4d, %s%4d %s%s\n",
                                "RUN= ", run_status, "MFP= ", MFP_status, "COMM= ",
                                comm_status, "RESTORE= ", restore_status,
                                "SAVE= ", save_status, "task= ", msg_task_id);
                    }

                    if (status != SN90_EOF)
```

```
                        printf("Error: sn90_next_task_status()\n", status);

                        returnStatus = 1;    /* force an exit  */
                        break;
                }
        case SN90_END_OF_SIMULATION:
                printf("taskstatus: END OF SIMULATION\n");
                returnStatus = 1;
                break;
        case SN90_NO_MESSAGE:
                break
        default:
                printf("taskstatus: MsgCode->%d Status->%d\n", msg_code, stat);
                status = sn90_ack_message();
                break;
    }
    return(returnStatus);
}
```

APPENDIX C - HARDWARE SETUP

INTRODUCTION

This appendix provides hardware setup procedures for the USM and MFP modules.

MFP MODULES IN SIMULATION MODE

The MFP module must be set to operate in the simulation special operation mode. When in simulation operation mode, the MFP front panel LEDs illuminate as shown in Figure C-1.

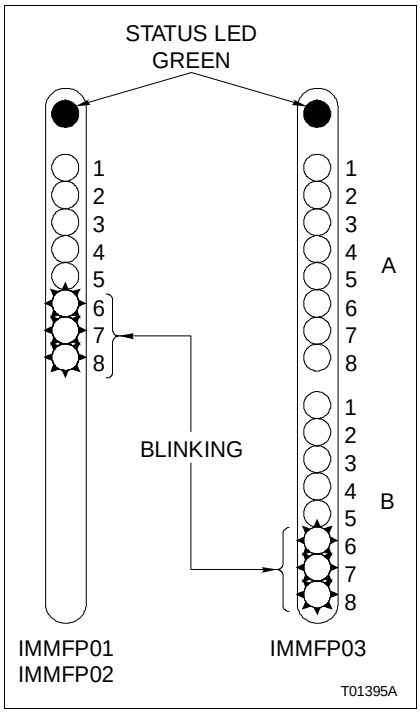


Figure C-1. Simulation Mode

To put the MFP module in simulation mode:

1. Pull out the MFP module.

2. Refer to the appropriate MFP module instruction and locate the respective special operations switch:

   IMMFP01/IMMPF02 - SW4

   IMMFP03/IMMFP03B - UMB1

3. Set switch poles 1, 5, and 8 to position 1.

4. Insert the MFP module and wait for the status LED to glow red and LEDs 1 through 6 to illuminate.

5. Remove the MFP module and set switch poles 1, 5, and 8 to position 0.

6. Insert the MFP module. The status LED should glow green and LEDs 6, 7, and 8 should blink, indicating that the MFP module is operating in simulation mode.

USM DIPSWITCH SETTINGS

Insure that dipswitch S3 on the USM main board is set for the ***boot from ROM in I/O space (default)*** boot option as described in Table 3-1 of the USM instruction.

Be sure to set the USM IP address switches S1 through S4 on the piggyback board correctly. In each byte, the most significant bit is in the right-most position (pole 8). Refer to Table C-1 for an example (IP address is 10.33.1.204).

Table C-1. USM IP Address

	S1	S2	S3	S4
	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8	1 2 3 4 5 6 7 8
Switch	0 1 0 1 0 0 0 0	1 0 0 0 0 1 0 0	1 0 0 0 0 0 0 0	0 0 1 1 0 0 1 1
Address	0 0 0 0 1 0 1 0	0 0 1 0 0 0 0 1	0 0 0 0 0 0 0 1	1 1 0 0 1 1 0 0

NOTE: 0 = CLOSED or ON, 1 = OPEN or OFF.

EXPANDER BUS DIPSHUNT

Insure that one connecting dipshunt exists so that the USM module and its MFP module can communicate over the expander bus. A common error is to omit the dipshunt or to have two dipshunts connecting to a single module.

Index

Index (continued)

Visit Elsag Bailey on the World Wide Web at http://www.ebpa.com

Our worldwide staff of professionals is ready to meet *your* needs for process automation.
For the location nearest you, please contact the appropriate regional office.

AMERICAS
29801 Euclid Avenue
Wickliffe, Ohio USA 44092
Telephone 1-216-585-8500
Telefax 1-216-585-8756

ASIA/PACIFIC
152 Beach Road
Gateway East #20-04
Singapore 189721
Telephone 65-391-0800
Telefax 65-292-9011

EUROPE, AFRICA, MIDDLE EAST
Via Puccini 2
16154 Genoa, Italy
Telephone 39-10-6582-943
Telefax 39-10-6582-941

GERMANY
Graefstrasse 97
D-60487 Frankfurt Main
Germany
Telephone 49-69-799-0
Telefax 49-69-799-2406

Form WBPEEUI350005B0 Litho in U.S.A. 897

Elsag Bailey
Process Automation