A Recommended Standard of the Joint Committee on the ATC
For Distribution by AASHTO, ITE and NEMA

ATC API Standard v02.06b

Application Programming Interface (API) Standard for the Advanced Transportation Controller (ATC)

September 21, 2007

Published by

American Association of State Highway and Transportation Officials (AASHTO)
444 North Capitol St., NW, Suite 249
Washington, DC 20001

Institute of Transportation Engineers (ITE)
1099 14th St. NW, Suite 300 West
Washington, DC 20005

National Electrical Manufacturers Association (NEMA)
1300 North 17th Street, Suite 1752
Rosslyn, VA 22209-3806

NOTICE

Joint NEMA, AASHTO and ITE Copyright and Advanced Transportation Controller (ATC) Application Programming Interface (API) Working Group

These materials are delivered "AS IS" without any warranties as to their use or performance.

AASHTO/ITE/NEMA AND THEIR SUPPLIERS DO NOT WARRANT THE PERFORMANCE OR RESULTS YOU MAY OBTAIN BY USING THESE MATERIALS. AASHTO/ITE/NEMA AND THEIR SUPPLIERS MAKE NO WARRANTIES, EXPRESSED OR IMPLIED, AS TO NON-INFRINGEMENT OF THIRD PARTY RIGHTS, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AASHTO, ITE, NEMA, OR THEIR SUPPLIERS BE LIABLE TO YOU OR ANY THIRD PARTY FOR ANY CLAIM OR FOR ANY CONSEQUENTIAL, INCIDENTAL, OR SPECIAL DAMAGES, INCLUDING ANY LOST PROFITS OR LOST SAVINGS ARISING FROM YOUR REPRODUCTION OR USE OF THESE MATERIALS, EVEN IF AN AASHTO, ITE, OR NEMA REPRESENTATIVE HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. Some states or jurisdictions do not allow the exclusion or limitation of incidental, consequential, or special damages, or exclusion of implied warranties, so the above limitations may not apply to you.

Use of these materials does not constitute an endorsement or affiliation by or between AASHTO, ITE, or NEMA and you, your company, or your products and services.

If you are not willing to accept the foregoing restrictions, you should immediately return these materials.

ATC is a trademark of NEMA/AASHTO/ITE.

REVISION NOTES

DATE	NOTE
09/26/06	Initial Working Group Draft (WGD) version 2.00 of the Advanced Transportation Controller (ATC) Application Programming Interface (API) Standard. Provides an outline/structure of the entire API Standard for approval by the Working Group (WG). Includes some of the detailed design (function descriptions) for the API.
11/16/06	WGD 2.01. Includes all changes based on comments to WGD 2.0. Includes all function descriptions.
12/03/06	User Comment Draft (UCD) 2.02. Includes suggested changes to WGD 2.01 made by the API WG prior to submittal to the ATC Joint Committee (JC).
04/11/07	WGD 2.03. Includes editorial changes based on comments to UCD 2.02 prior to WG adjudication.
04/15/07	WGD 2.04. Includes all changes based on the adjudicated user comments to version 2.02 except time functions.
04/23/07	Proposed Recommended Standard 2.05. Includes new time library functions and revisions to the FPUI library.
05/15/07	Proposed Recommended Standard 2.06. Added explanatory text and minor corrections to the Time of Day Section 4.3.1.
05/31/07	Recommended Standard 2.06a.
09/21/07	Jointly Approved Standard 2.06b with only editorial corrections to 2.06a.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

This section provides an introduction for this document. It includes sections on "Purpose;" "Scope;" "Definitions, Acronyms, and Abbreviations;" "References;" and "Overview."

1.1 Purpose

The Advanced Transportation Controller (ATC) Standards are intended to provide an open architecture hardware and software platform that can support a wide variety of Intelligent Transportation Systems (ITS) applications including traffic management, safety, security and other applications. The ATC Standards are being developed and maintained under the direction of the ATC Joint Committee (JC) which is made up of representatives from the American Association of State Highway and Transportation Officials (AASHTO), the Institute of Transportation Engineers (ITE) and the National Electrical Manufacturers Association (NEMA).

This document defines a software interface for application programs intended to operate on ATC controller units. It has been prepared by the ATC Application Programming Interface (API) Working Group (WG), a technical subcommittee of the ATC JC. It establishes a common understanding of the user needs, requirements and specification of the interface for:

a) The local, state and federal transportation agencies who specify ATC equipment;

b) The software developers, consultants and manufacturers who develop application programs for ATC equipment; and

c) The public who benefit in the application programs that run on ATC equipment and directly or indirectly pay for these products.

1.2 Scope

The ATC Controller Standard defines a controller that can grow with technology. It is made up of a central processing unit (CPU), an operating system (O/S), memory, external and internal interfaces and other associated hardware necessary to create an embedded transportation computing platform. The goal of the interface described in this standard is to define a software platform that, when combined with the ATC O/S, forms a universal interface for application programs. This interface allows application programs to be written so that they may run on any ATC controller unit regardless of the manufacturer. It also defines a software environment that allows multiple application

programs to be interoperable on a single controller unit by sharing the fixed resources of the controller. The API Standard specifies the interface. Software developed in compliance with the API Standard is known as the ATC Application Programming Interface (API).

1.3 Definitions, Acronyms and Abbreviations

TERM	DEFINITION
170	A traffic control device that meets one of the California Department of Transportation (Caltrans) standards for Model 170 traffic control devices.
2070	A traffic control device that meets one of the California Department of Transportation (Caltrans) standards for Model 2070 traffic control devices or one of the standards for the ATC 2070 traffic control devices.
AASHTO	American Association of State Highway and Transportation Officials
API	Application Programming Interface. In this standard, when API is used by itself as a noun it refers to the software that is compliant to the ATC API Standard.
API Managers	API software that manages an ATC resource for use by concurrently running application programs.
API Utilities	API software not included in the API Managers that is used for configuration purposes.
Application Program	Any program designed to perform a specific function directly for the user or, in some cases, for another application program. Examples of application programs include word processors, database programs, Web browsers and traffic control programs. Application programs use the services of a computer's O/S and other supporting programs such as an application programming interface.
ASCII	American Standard Code for Information Interchange. A standard character-coding scheme used by most computers to display letters, digits and special characters. See ANSI-X3.4-1986(R 1997).
ATC	Advanced Transportation Controller
ATC Device Drivers	Low-level software not included in standard Linux distributions that is necessary for ATC-specific devices to operate in a Linux O/S environment.
Aux Switch	A physical “auxiliary” switch on the Front Panel of ATC 2070 controllers that may be used by application programs.

TERM	DEFINITION
BIU	Bus Interface Unit. A transportation cabinet device which is used for communications within some cabinet systems.
Board Support Package	Software usually provided by processor board manufacturers which provides a consistent software interface for the unique architecture of the board. In the case of the ATC, the Board Support Package also includes the O/S.
BSP	See Board Support Package.
CMU	Cabinet Monitor Unit. A transportation cabinet device which monitors the operational status of some cabinet systems.
CPU	Central Processing Unit. A programmable logic device that performs the instruction, logic and mathematical processing in a computer.
Device Driver	A software routine that links a peripheral device to the operating system. It acts like a translator between a device and the application programs that use it.
DRAM	Dynamic Random Access Memory
DST	Daylight Saving Time
EEPROM	Electrically Erasable, Programmable, Read-Only Memory. EEPROMs differ from DRAMs in that the memory is saved even if electrical power is lost.
ELF	Executable and Linking Format. A software library format.
Epoch	An origin of time measurement commonly used in computers. It is midnight UTC of January 1, 1970.
FAT	File Allocation Table. A Microsoft Windows file system format.
FHWA	Federal Highway Administration
FIFO	First In/First Out. A method of storage in which the data stored for the longest time will be retrieved first.
FIO	Field Input and Output
FSA	Failed State Action
I/O	Input/Output
IEC	International Engineering Consortium
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
ITE	Institute of Transportation Engineers
ITS	Intelligent Transportation Systems
JC	Joint Committee
kbps	Kilobits per second (thousands of bits per second)

TERM	DEFINITION
LED	Light Emitting Diode. A display technology that uses a semiconductor diode that emits light when charged.
Linux Device Drivers	Low-level software that is freely available in the Linux community for use with common hardware components operating in a standard fashion.
Linux Kernel	The Unix-like operating system kernel that was begun by Linus Torvalds in 1991. The Linux Kernel provides general O/S functionality. This includes functions for things typical in any computer system such as file I/O, serial I/O, interprocess communication and process scheduling. It also includes Linux utility functions necessary to run programs such as shell scripts and console commands. It is generally available as open source (free to the public). The Linux Kernel referenced in this standard is defined in the ATC Controller Standard, Section 2.2.5, Annex A and Annex B.
Mb/s	Mega/Million Bits per Second
MMU	Malfunction Management Unit. A transportation cabinet device which monitors the operational status of some cabinet systems.
ms	Millisecond. One thousandth (10^{-3}) of a second.
NEMA	National Electrical Manufacturers Association
NEMA Controller	A traffic control device that meets one of the NEMA standards for traffic control devices.
Operational User	A technician or transportation engineer who uses the controller to perform its operational tasks.
O/S	Operating System
PCB	Printed Circuit Board
Process	A process is an instance of a program running in a computer. In Linux, a process is started when a program is initiated (either by a user entering a shell command or by another program). A process is a running program in which a particular set of data and unique process identifier (ID) is associated so that the process can be managed by the operating system.
RAM	Random Access Memory
RTC	Real-Time Clock
SDD	Software Design Document
SDLC	Synchronous Data Link Control. A communication protocol used in some transportation cabinet systems.
SDO	Standards Development Organization

TERM	DEFINITION
SIU	Serial Interface Unit. A transportation cabinet device which is used for communications within some cabinet systems.
SRAM	Static Random Access Memory
SW	Software
TOD	Time of Day
TTL	Transistor Transistor Logic
WG	Working Group
USDOT	United States Department of Transportation
USB	Universal Serial Bus. An external peripheral interface standard for communication between a computer and external peripherals.
User Developer	A software developer that designs and develops programs for controllers.
UTC	Coordinated Universal Time. A high-precision atomic time standard. Time zones around the world are expressed as positive or negative offsets from UTC. As the zero-point reference, UTC is also referred to as Zulu time (Z). UTC is often referred to as Greenwich Mean Time when describing time zones.

1.4 References

The documents referenced by this standard are listed below.

"ATC Controller Standard Revision v5.2b," ATC JC, 26 June 2006. Available from the Institute of Transportation Engineers.

"ATC Standard for the Type 2070 Controller v01.05," ATC JC, 29 March 2001. Available from the Institute of Transportation Engineers.

"GNU Coding Standards," 8 May 2006. Available from Free Software Foundation, Inc.

"IEEE Recommended Practice for Software Design Descriptions," IEEE Std 1016-1998. Available from the Institute of Electrical and Electronics Engineers.

"IEEE Recommended Practice for Software Requirements Specifications," IEEE Std 830-1998. Available from the Institute of Electrical and Electronics Engineers.

"Intelligent Transportation System (ITS) Standard Specification for Roadside Cabinets v01.02.17b," ATC JC, 16 November 2006. Available from the Institute of Transportation Engineers.

"ISO/IEC 9899:1999 Programming Language C." Available from the International Organization for Standardization (ISO).

"NEMA Standards Publication TS 2-2003 v02.06 Traffic Controller Assemblies with NTCIP Requirements." Available from the National Electrical Manufacturers Association.

1.5 Overview

This standard is made up of four sections, appendixes and an index. Section 1, "Introduction," provides an overview of the entire document. Section 2, "Overall Description," provides background information and the user needs for the requirements defined in the subsequent section. User needs within the subsections of Section 2 are identified by numbers in square brackets (Ex. "Stated user need … [1]."). Section 3, "Specific Requirements," defines the requirements that must satisfied by the ATC API. Requirements within the subsections of Section 3 are identified by numbers in square brackets (Ex. "Stated requirement … [1]."). Section 4, "Application Programming Interface," specifies the ATC API. The user needs, the requirements and the API functions that satisfy the requirements are crossed referenced in a traceability matrix in Appendix A.

2 OVERALL DESCRIPTION

This section provides the description of user needs for the ATC API. It includes sections on “Product Perspective;” “Product Features;” “User Characteristics;” “Constraints;” “Assumptions and Dependencies;” and “Apportioning of Requirements.” User needs within a section are identified by numbers in square brackets (Ex. “Stated user need … [1].”). Each user need identified is also included in a traceability matrix in Appendix A. The terms “function” or “functions” used in this section refer to software functions or software function calls. If a general functionality or capability is intended, an effort was made to use other terms.

2.1 Product Perspective

The ATC API Standard is one of four standards efforts under the direction of the ATC JC. It is the intent of the ATC JC to create a set of standards that fulfill the computing needs for transportation applications of today and also provide an architecture that can grow to help meet the transportation needs of the future. Other standards within the ATC program include the ATC/2070 Standard, the ATC Controller Standard and the Intelligent Transportation Systems (ITS) Cabinet Standard (see Section 1.4 for complete names and versions of these documents).

Historically, application programs written by one manufacturer do not run on equipment from another manufacturer. This is not just on equipment as different as 170 and NEMA controllers but even between NEMA controllers from different manufacturers. While there have been software portability successes within the 170 markets, the hardware specifications of these architectures do not lend themselves to adopt the latest hardware or software technologies nor do they create the market that is necessary to support the broad range of application programs required in the future. See Figure 1.

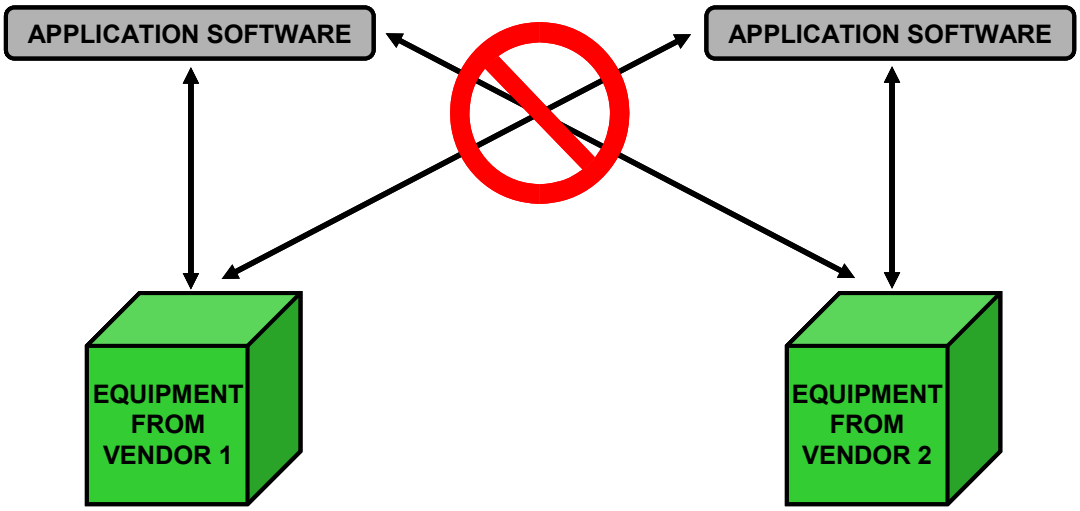

Figure 1. Non-portable software traditionally used in transportation industry.

When combined with the ATC operating system, the API forms an open architecture software (SW) platform that acts as a universal interface between application programs and the ATC controller units [1]. Open architecture, as used, means that the function specifications are documented and available to everyone. This includes the ATC operating system and the API SW. It should be noted that application programs and proprietary loadable device driver modules supplied with the ATC controller unit may not be considered open source. The controller unit includes the central processing unit (CPU), the operating system (O/S), memory, external and internal interfaces and all other associated hardware that is defined by the ATC Controller Standard. The O/S and API "abstract" the application program from the ATC hardware, allowing application programs to be written that can be made to operate on any ATC (regardless of manufacturer). In older controller architectures, the source code of application programs would require considerable modification and, in some cases, to be completely rewritten to run on a different vendor's platform. The API, when used with the ATC O/S, facilitates portability by requiring only modest efforts on the part of the developer such as recompiling and linking source code for a particular processor [2]. See Figure 2.

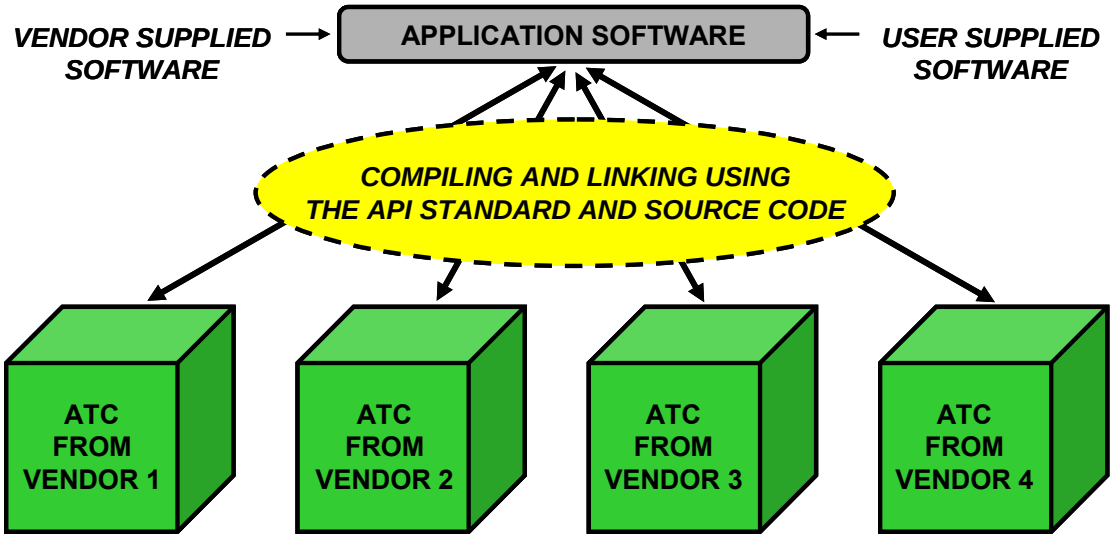

Figure 2. Application program portability using the API Standard.

The ATC Controller Standard specifies a controller architecture where the computational components reside on a printed circuit board (PCB), called the "Engine Board," with standardized connectors and pinout. The Engine Board contains the following items:

a) CPU;
b) Linux O/S and Device Drivers;
c) Non-Volatile (Flash) Memory;
d) Dynamic and Static RAM (DRAM and SRAM);
e) Real-Time Clock (RTC);
f) Two Ethernet ports;
g) One Universal Serial Bus (USB) port that is used for a portable memory device; and
h) Eight serial ports (some are designated for special interfaces and others general purpose).

The Engine Board plugs into a "Host Module" that supplies power and physical connection to the I/O devices of the controller. While the interface to the Engine Board is completely specified, the Host Module may be of various shapes and sizes to accommodate controllers of various designs. Figure 3 shows how the Engine Board can be used to create ATC controllers that work within different families of traffic controller equipment. This concept also allows more powerful Engine Boards to be deployed in the future without changing the overall controller and cabinet architecture.

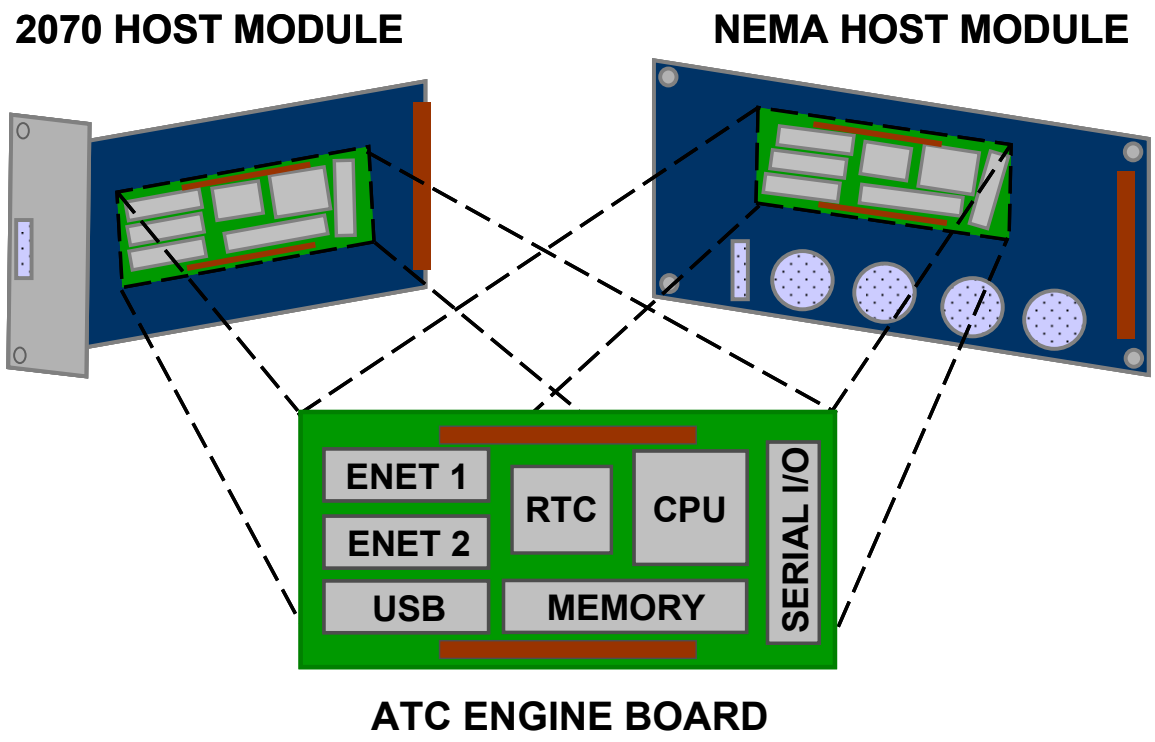

Figure 3. ATC Engine Board being used to support different families of controllers.

The ATC Controller Standard specifies a minimum level of real-time processing capability for the Engine Board. It also specifies the minimum physical and communication requirements for the Host Module. Some of the communication ports of the Engine Board are general-purpose, intended to be used via the O/S as needed by application programs. Other ports are specific to traffic controller devices. These ports are managed by the API and are intended to be shared across concurrently running application programs. Figure 4 shows the Engine Board communication ports and their intended use. As shown, the API must provide management functions for the Front Panel and the Field I/O Devices [3]. Using the API, future advances in processing power can be applied to Engine Boards, installed into existing ATC controllers and still operate the application programs of the transportation system (recompiling may be required as shown in Figure 2).

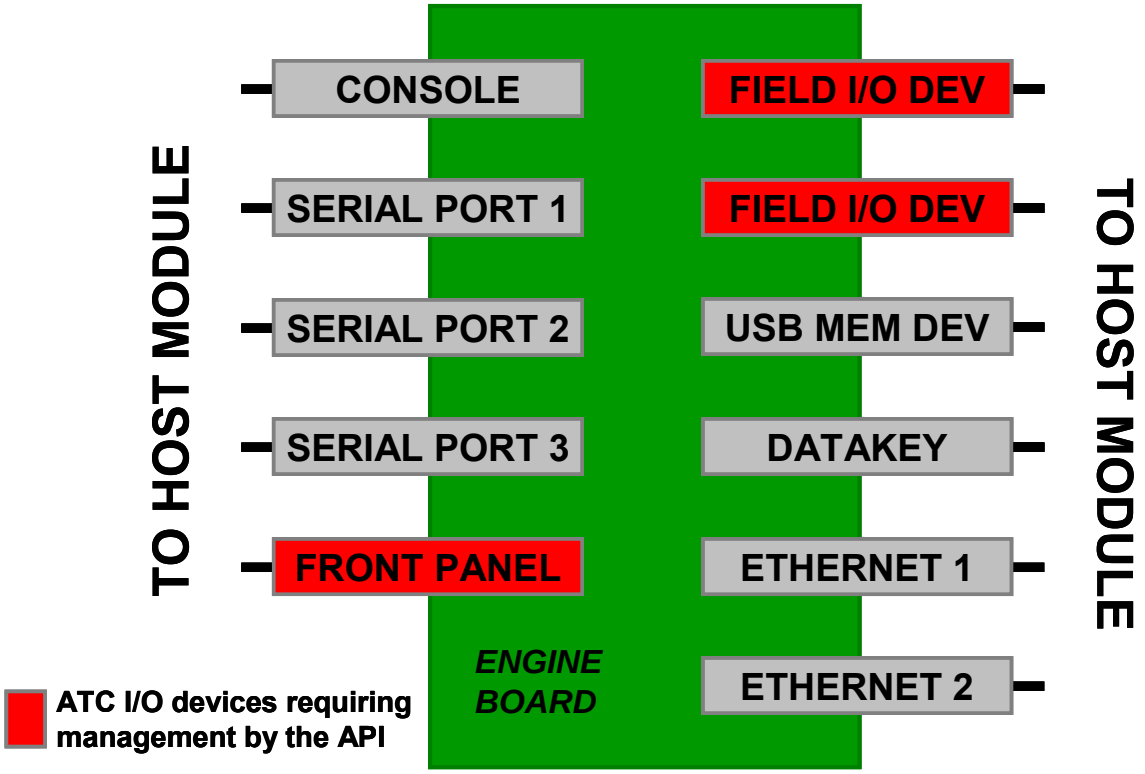

Figure 4. ATC Engine Board I/O supported by the API.

Figure 5 illustrates the organization and layered architecture of the ATC software. The "Linux O/S and Device Drivers" reflects a specification of the Linux operating system defined in the ATC Board Support Package (BSP) (see ATC Controller Standard, Section 2.2.5, Annex A and Annex B). This includes functions for things typical in any computer system such as file I/O, serial I/O, interprocess communication and process scheduling. It also includes the specification of the device drivers necessary for the Linux O/S to operate on the ATC hardware. "API" refers to the software specified in Section 4 of this standard. As shown in Figure 5, both users and application programs use the API to interface to ATC controller units [4].

The division of the ATC software into layers helps to insure consistent behavior of the software environment between ATC architectures and also provides a migration path to new ATCs in the future. The relationship between the Hardware Layer and ATC BSP Layer is maintained, for the most part, by the Linux operating system community of users. Linux source code licenses are free to the public, and there are strong market incentives for Linux users to maintain the Linux standard and insure consistent functionality of the Linux commands for the operating system. The relationship between the ATC BSP Layer and the API Software Layer is maintained by the transportation community. Functions in the API Software Layer access the ATC unit through the

functions in the ATC BSP Layer [5]. If programs written for the Application Layer only reference the ATC unit through the functions specified in the API Software Layer and ATC BSP Layer, they will be able to operate on any ATC provided the source code is recompiled for the target ATC's processor.

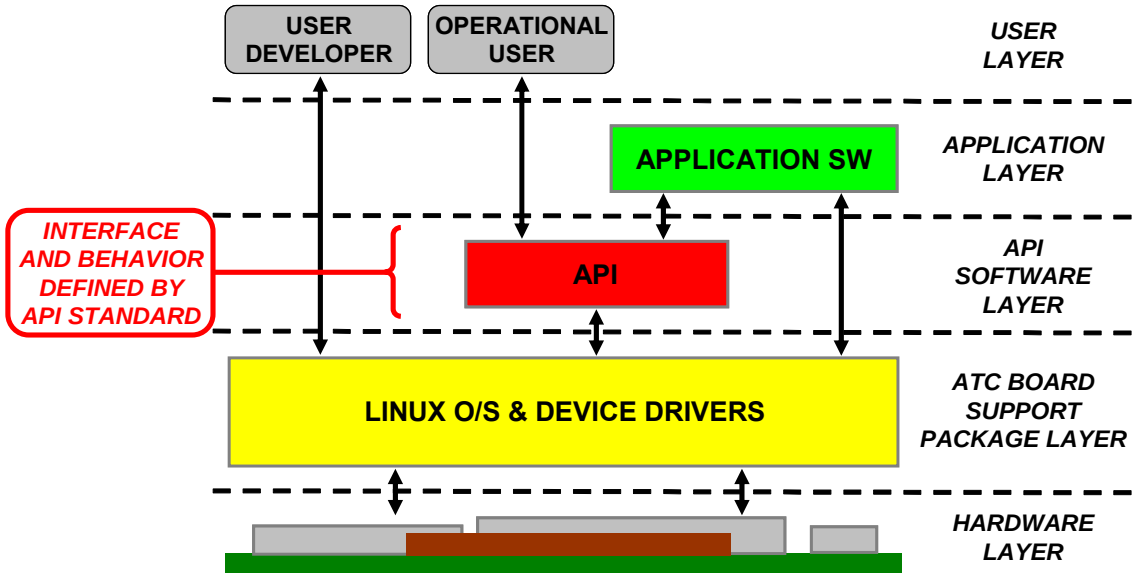

Figure 5. ATC software layered organization.

2.1.1 System Interfaces

There are no system interfaces addressed by this standard. A system, as intended here, is a collection of related hardware and software products whose behavior satisfies an operational need. By this definition, a controller unit and its operational software form a system. The API is a component of this system but does not directly address any system level interfaces.

2.1.2 User Interfaces

The ATC Controller Standard defines a Front Panel interface for Operational Users to interact with the application programs that are the primary tasks of the controller unit within the transportation system. Examples of these programs are intersection control, field management stations, ramp metering, radiation detection, etc. Historically, a controller unit would support only one application program at a time and application program developers could assume they had unimpeded and constant access to a controller unit's Front Panel. ATC controller units support the use of multiple application

programs running concurrently. This requires the API to support a user interface that provides Operational Users with the ability to select from a set of active application programs on the ATC controller unit and to interact with the programs individually [1]. The ATC Controller Standard describes the physical characteristics and the character set to be supported [2].

The API must also provide C functions that allow application programs to access and control the Front Panel Interface programmatically [3]. User Developers reference these functions in the source code of their application programs. A Console Interface is provided to allow User Developers to load programs and maintain both application programs and the Linux environment itself. This interface is defined in the ATC Controller Standard and is outside the scope of the API Standard as shown in Figure 5.

2.1.3 Hardware Interfaces

The API manages the resources of the ATC controller unit which are intended to be shared between concurrently operating application programs and whose functionality is not inherently supported in Linux (or by the device drivers created for the ATC). These managed resources include the Front Panel (see Section 2.1.2 User Interfaces) and the Field I/O Devices [1]. The API does not communicate directly to these ATC resources but through the Linux operating system and associated device drivers provided with the ATC controller unit [2]. The hardware requirements for these interfaces are described in the ATC Controller Standard.

2.1.4 Software Interfaces

There are no software interfaces required other than the Linux O/S and BSP defined in the ATC Controller Standard.

2.1.5 Communications Interfaces

The ATC Controller Standard states that ATC controller units have three or four general purpose EIA 485 serial ports (one of the ports can be general purpose or designated specifically for Field I/O Devices, see ATC Controller Standard, Section 5.4.3) and two Ethernet ports used for communications purposes. The ATC controller unit provides access to these communications interfaces via the Linux O/S and Device Drivers.

2.1.6 Memory Constraints

The API must operate effectively within the memory constraints defined for ATC controller units in the ATC Controller Standard [1].

2.1.7 Operations

See Section 2.1.2 User Interfaces.

2.1.8 Site Adaptation Requirements

There are no site adaptation requirements.

2.2 Product Features

The API Software Layer and ATC BSP Layer discussed in the Section 2.1 are further divided into five functional areas as shown in Figure 6. The ATC Controller and API Standards specify the requirements for these areas providing both a programmatic interface for application programs and a user interface for operational use of the application programs.

- The “Linux Kernel” provides general O/S functionality. This includes functions for things typical in any computer system such as file I/O, serial I/O, interprocess communication and process scheduling. It also includes Linux utility functions necessary to run programs such as shell scripts and console commands.

- “Linux Device Drivers” refers to low-level software that is freely available in the Linux community for use with common hardware components operating in a standard fashion.

- “ATC Device Drivers” refers to low-level software not included in standard Linux distributions that is necessary for ATC-specific devices to operate in a Linux O/S environment.

- “API Managers” refers to the functions that are intended to manage the operation of the ATC hardware interfaces listed in Section 2.1.3.

- “API Utilities” refers to the functions not included in the API Managers that are used for configuration purposes.

The arrows in Figure 6 show the logical interfaces of the functional areas. Operational Users must interface with application programs through the API to perform the primary

tasks assigned to the controller unit [1]. Utility functions are necessary for configuration purposes [2]. Application programs interface to the Linux Kernel for those functions common to operating systems. Application programs interface to the API Manager Functions, API Utilities and the Linux Kernel for access to the ATC-specific devices. The API Manager Functions and the API Utilities access the ATC controller hardware through the Linux Kernel [3]. The Linux Kernel and Linux Device Drivers are defined in the ATC Controller Standard, Annex A, as a profile (selected subset) of commonly available Linux O/S software. The specifications for the ATC Device Drivers are defined in the ATC Controller Standard, Annex B. The ATC Device Driver software is to be developed by manufacturers as appropriate for the architecture of their ATC hardware. The software for the API Manager Functions and the API Utilities is to be developed in response to this standard.

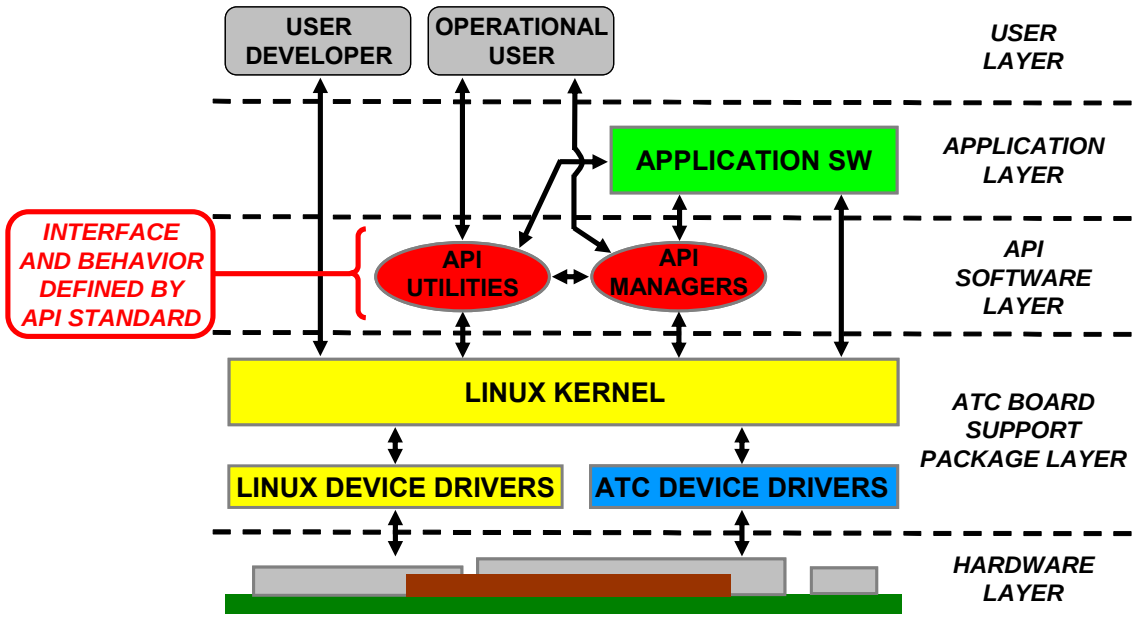

Figure 6. ATC API functional areas.

Functions in the area Linux Kernel include those that are called from within software programs and those that are invoked by the user via the console interface. The Linux Kernel is not considered a part of the API but is described here to identify assumptions made in the API Standard.

a) The Linux Kernel provides software signaling services including creating/deleting, scheduling and sending/receiving software events. These services must include single occurrence, multiple occurrence and signal by date.

b) The Linux Kernel provides process management services that allow processes to be spawned from other processes, process priority management, waiting on a process, process sleep, process termination, retrieving process information and exiting a process.

c) The Linux Kernel provides shared memory capabilities including allocation, access control and deletion.

d) The Linux Kernel provides semaphore and mutex synchronization services including initialization/termination, acquisition/release, increment/decrement and waiting.

e) The Linux Kernel provides serial service functions including opening/closing a port, reading/writing, setting/getting properties, status checking and signal notification.

f) The Linux Kernel provides pipe service functions including opening/closing a pipe, reading/writing, setting/getting properties, status checking and signal notification.

g) The Linux Kernel provides system time functions including getting/setting the system time.

h) The Linux Kernel provides capabilities to set/get system global variables.

i) The Linux Kernel provides network configuration capabilities with TCP/IP and UDP/IP socket communication services including open, close, read and write.

j) The Linux Kernel provides basic shell user interface and scripting mechanism via a console port.

The functions in Linux Device Drivers support the Linux O/S functions used in the Linux Kernel that operate on common hardware components. The Linux Device Drivers are specified in the ATC Controller Standard, Annex A.

Functions in the area ATC Device Drivers are driver programs (software) that are necessary to interface to the ATC-specific hardware devices. A custom driver may have to be developed if a suitable driver is not available within the Linux open source community. These drivers are not considered a part of the API, but they are described here to identify assumptions made in the API Standard.

a) Datakey Interface Driver. This device driver must support all of the allowable key sizes as specified in the ATC Controller Standard. The

device driver must allow multiple application programs to access the data key as a file system.

b) Front Panel Interface Driver. This device driver provides a serial interface to the controller's Front Panel as described in the ATC Controller Standard. This device driver is used by the API Front Panel Manager exclusively.

c) Serial I/O Interface Drivers. These device drivers provide synchronous and asynchronous serial ports as described in the ATC Controller Standard.

d) FLASH Interface Driver. This device driver provides a flash file system interface as described in the ATC Controller Standard. The device driver must allow multiple application programs to access the file system concurrently.

e) SRAM Interface Driver. This device driver provides a file system interface as described in the ATC Controller Standard.

f) USB Interface Driver. This device driver provides FAT File I/O services for the USB Portable Memory Device.

g) Real-time Clock Interface Driver. This device driver provides access to the battery-backed, real-time clock on the ATC Engine Board.

h) Miscellaneous Parallel I/O Drivers. These device drivers provide access to specific I/O pins such as POWERDOWN, CPU_RESET, CPU_ACTIVE and DKEY_PRESENT as described in the ATC Controller Standard. The CPU_ACTIVE device driver controls the CPU ACTIVE LED Indicator. Access to the CPU_ACTIVE device driver is managed by the API (see Section 3.3.1.2).

i) Serial Peripheral Interface Drivers (SPI). These device drivers provide an interface to the EEPROM and also work with the Datakey as described in the ATC Controller Standard.

j) Ethernet Interface Driver. This driver provides an interface to the Ethernet ports of the Engine Board as described by the ATC Controller Standard.

Functions in the area API Managers provide the interface to and management of the shared resources of the ATC controller not sufficiently covered by functions inherent in Linux or provided for in any custom driver.

a) Front Panel Manager. This set of functions provides a user interface which allows programs running concurrently on an ATC controller to share the controller's Front Panel through the dedicated Front Panel Port of the ATC Engine Board [4].

b) Field I/O Manager. This set of functions gives concurrently running programs the capability to communicate with Field I/O Devices connected to an ATC controller through the dedicated Field I/O ports of the ATC Engine Board [5].

Functions in the area API Utilities provide the Operational User tools in which to configure and maintain the API installed on an ATC. While some of these functions are based on the implementation of the managers and outside of this standard, there is a need for Operational Users to have access to the API version and configuration information [6].

2.3 User Characteristics

There are two types of users of the API. The "Operational User" is a technician or transportation engineer that uses the controller to perform its operational tasks. If they are engineers, they typically have civil or mechanical engineering backgrounds. The "User Developer" is a software developer that designs and develops programs for controllers. These individuals typically have software, computer, or electrical engineering backgrounds.

2.4 Constraints

The API must operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard [1]. The API function calls must be specified using the C programming language as described by "ISO/IEC 9899:1999," commonly referred to as the C99 Standard [2]. The operational look and feel of user interfaces developed for the API should be consistent with each other [3]. If API functions have a similar operation to existing Linux functions, they should have a similar name and argument style to those functions to the extent possible without causing compilation issues [4]. The API functions should use consistent naming conventions, argument styles and return values [5]. The API should be available to users as a library that is loadable via a start-up script [6].

2.5 Assumptions and Dependencies

It is assumed that the API will operate on an ATC controller unit as defined by the ATC Controller Standard. It is assumed that different software implementations of this standard may require User Utilities not specifically stated in this standard. If this is the case, these User Utilities should be made available to Operational Users through software release notes or other means.

2.6 Apportioning of Requirements

There is no apportioning of the requirements of this standard.

3 SPECIFIC REQUIREMENTS

This section specifies the requirements for the ATC API. It includes sections on “API Manager Requirements;” “API Utility Requirements;” “Performance Requirements;” “Design Constraints;” “Software System Attributes;” and “Other Requirements.” Requirements within a section are identified by numbers in square brackets (Ex. “Stated requirement … [1].”). Each requirement identified is also included in a traceability matrix in Appendix A. The terms “function” or “functions” used in this section refer to software functions or software function calls. If a general functionality or capability is intended, an effort was made to use other terms.

3.1 API Manager Requirements

This section specifies the requirements for the ATC API Front Panel Manager and Field I/O Manager.

3.1.1 Front Panel Manager Requirements

The API shall provide a text-based user interface capability to allow application programs running concurrently on an ATC controller unit to share the controller’s Front Panel display [1]. The API shall provide up to 16 virtual display screens (referred to as “windows”) that can be used by application programs as their user interface display [2]. The display size of the windows shall be equal to physical display size (lines x characters) of the controller’s Front Panel display (if one exists) [3]. The display size of the windows shall have a minimum size of 4 lines x 40 characters and a maximum size of 24 lines x 80 characters [4]. If no physical display exists, the API shall operate as if it has a display with a size of 8 lines x 40 characters [5]. Only one window shall be displayed at a time on the Front Panel display [6]. When a window is displayed, the API shall display the character representation of the window on the Front Panel display (if one exists) [7]. The application program associated with the window displayed shall receive the characters input from the Front Panel input device (Ex. keyboard or keypad) [8]. When an application window is displayed, it is said to have “focus.” The window which has focus when the controller is powered up is called the “default window.” The API shall support the display character set as defined in the ATC Controller Standard, Section 7.1.4 [9]. Screen attributes described by the ATC Controller Standard, Section 7.1.4, shall be maintained for each window independently [10]. Each window shall have separate input and output buffers unique from other windows [11].

Guidance: An ATC Controller Standard requires that an ATC controller unit support various sizes of physical Front Panel displays and operate in the absence of a Front Panel display.

3.1.1.1 Front Panel Manager Window Requirements

The API shall provide a window selection screen called the Front Panel Manager Window from which Operational Users may select a window to have focus [1]. Application names associated with each window shall be listed [2]. The application names shall be limited to 16 characters [3]. If there is no application program associated with a window, the window number shall be listed with a blank application name [4]. The default Front Panel Manager Window size shall be 8 lines x 40 characters with the format as shown in Figure 7 [5].

	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0	1	2	3	4	5	6	7	8	9	0
1											F	R	O	N	T		P	A	N	E	L		M	A	N	A	G	E	R											
2	S	E	L	E	C	T		W	I	N	D	O	W	:		0	-	F				S	E	T		D	E	F	A	U	L	T	:		*	,	0	-	F	
3	0	*	*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*0*			1		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*1*		
4	2		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*2*			3		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*3*		
5	4		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*4*			5		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*5*		
6	6		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*6*			7		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*7*		
7	8		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*8*			9		*a*	*p*	*p*	*l*	*i*	*c*	*a*	*t*	*i*	*o*	*n*	*n*	*a*	*m*	*e*	*9*		
8	[	U	P	/	D	N		A	R	R	O	W	]								[	C	O	N	F	I	G		I	N	F	O	-		N	E	X	T	]

Shaded areas ▢ are row and column headings. They are not a part of the actual display.
***applicationname#* refers to the application name associated with the window.**
If there is no task assigned to a window, it will be blank.
*** indicates the window displayed when the controller unit is started (the "default window").**

Figure 7. ATC Front Panel Manager Window.

Key sequences used in the requirements below use the following conventions:
- Curled brackets ({}) are used to delimit a key sequence.
- Square brackets ([]) signify a key from a range or set such as [0-F] or [A, B, C].
- A range of [0-F] signifies the hexadecimal characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F.
- Angle brackets (<>) are used to identify a special character such as <ESC> or <NEXT>.
- A comma (,) used in a key sequence is not entered but is used to delimit keys.

a) If the Operational User has not set the default window, the Front Panel Manager Window shall be the default window [6].

b) The default window shall be settable by the Operational User from the Front Panel Manager Window by pressing {*,[0-F],<ENT>} [7].

c) The Operational User shall be capable of setting the default window to the Front Panel Manager Window by pressing {*,<ENT>} from the Front Panel Manager Window [8].

d) The default window shall be designated by a star "*" character next to the window number [9].

e) The Operational User shall be able to put the Front Panel Manager Window in focus by pressing {**,<ESC>} from the keypad on the controller's Front Panel regardless of the application program in operation [10]. The Operational User shall be able to enter {**} by pressing an asterisk (*) twice within a 1.0 second time period [11]. If the {**} sequence is not completed within the 1.0 second time period or if the {**} sequence is not followed by <ESC> character within a 1.0 second time period, then the characters shall be interpreted as individual "*" characters [12].

f) The Operational User shall have the capability to put a window in focus that is assigned to an application program by pressing {[0-F]} from the Front Panel Manager Window [13].

g) The only possible window selections for focus from the Front Panel Manager Window shall be itself, the ATC Configuration Window (see Section 3.2), or a window assigned to an application program [14].

h) If the Front Panel Manager Window is the default window, no asterisk shall be displayed next to any application name in the Front Panel Manager Window [15].

i) The Operational User shall be able to put the ATC Configuration Window in focus by pressing {<NEXT>} in the Front Panel Manager Window [16].

j) The top two lines and bottom line of the Front Panel Manager Window shall be fixed as shown in Figure 7 [17].

k) The number of lines between the second line and bottom lines used for displaying window names shall vary according to the size of the ATC display [18].

l) The Operational User shall be able to scroll up and down the names of the windows in the Front Panel Manager Window one line at a time using the up and down arrow keys of the controller keypad [19].

3.1.1.2 Front Panel Manager Software Interface Requirements

The API provides a software interface enabling application programs to manipulate windows as described below.

a) The API shall provide a function to return the dimensions of a window in terms of number of lines and number of columns [1].

b) The API shall provide a function to open a window and register a name for display on the Front Panel Manager Window [2]. An application program shall be able to open multiple windows providing the windows resources are available [3].

c) The API shall provide the ability for an application program to reserve exclusive access to the Aux Switch (see ATC Controller Standard, Section 7.1.4) [4]. An application program that has reserved exclusive access to the AUX Switch shall maintain exclusive access to the switch even if the application program has no window in focus [5].

d) The API shall provide a function to close a window and release the resource for other application programs [6].

e) The API shall provide a function or set of functions to set the attributes of a Front Panel display as described in the ATC Controller Standard, Section 7.1.4 [7].

f) The API shall provide a function or set of functions to return the attributes of a Front Panel display as described in the ATC Controller Standard, Section 7.1.4 [8].

g) The API shall provide a function that is used to determine if there is data in the input buffer of a window [9].

h) The API shall provide a function to read a queued character or key code from the input buffer of a window [10].

i) The API shall provide a function to write a character to the current cursor position of a window [11].

j) The API shall provide a function to write a character to a window at a position defined by column and line number [12].

k) The API shall provide a function to write a string to a window at the current cursor position [13].

l) The API shall provide a function to write a string to a window at a starting position defined by column number and line number [14].

m) The API shall provide a function to write a buffer of characters to a window at the current cursor position [15].

n) The API shall provide a function to write a buffer of characters to a window at a starting position defined by column number and line number [16].

o) The API shall provide a function to set the cursor position of a window defined by column and line number [17].

p) The API shall provide a function to return the cursor position of the window defined by column and line number [18].

q) If a window was registered with access to the Aux Switch, the API shall provide a function to return its status [19].

r) The API shall provide a function to compose special characters as described by the ATC Controller Standard, Section 7.1.4 [20].

s) The API shall support the display of a composed character in the same manner as any other valid character [21].

t) The API shall provide a function to clear a window that operates on a window whether it is in or out of focus [22].

u) The API shall provide a function to refresh a window that operates on a window whether it is in or out of focus [23].

v) The bell of the controller's Front Panel shall be activated only if a bell character, ^G (hex value 07), is sent from an application program which has a window that has focus [24]. If a bell character is sent from an application program that does not have a window that has focus, the bell character shall be ignored by the API [25].

w) The API shall allow application programs to illuminate or extinguish the backlight of the ATC controller's display if the command is received through a window that is in focus [26].

x) Display configuration and inquiry command codes (escape sequences) specified in the ATC Controller Standard, Section 7.1.4, shall be supported as separate functions in the API [27].

y) Application programs shall be able to interpret all ATC controller keys as individual key codes [28]. The escape sequences representing keys that

do not have standard ASCII character codes on an ATC controller shall be mapped to specific character codes in the API as shown in Table 1 [29].

z) The ATC Controller Standard, Section 7.1.4, describes a graphics interface to the Front Panel's display. The API shall support the operation of the graphics commands on a window only if that window is in focus [30]. If application programs use graphics on a window, the API shall not redisplay these graphics when a window is refreshed or goes out/in focus [31].

Guidance: Application programs may store graphics commands used on a window that is out of focus and execute them once the window has gone into focus.

aa) The API shall provide an asynchronous notification to alert programs when their associated windows go in and out of focus [32].

bb) The API shall provide a function which application programs may use to determine if their window is in focus [33].

cc) The API shall provide a method to allow application programs to indicate that a window desires focus from the Operational User [34]. This method shall cause the Front Panel backlight to flash and the window name in the Front Panel Manager Window to blink [35]. The window name blinking shall cease once the indicated window receives focus [36]. The backlight flashing shall cease when all windows requesting focus have been given focus [37].

dd) The API shall provide a mechanism to allow application programs to detect the presence or absence of a front panel [38]. The API shall recognize the presence or absence of the front panel in 5 seconds [39]. The API shall provide an asynchronous notification to alert application programs of a change in the presence or absence of the front panel [40].

Guidance: It is suggested that a Request Cursor Position command or similar technique be used to determine the presence of a front panel.

Guidance: If a front panel is removed and replaced (or installed and removed) within the 5 second period specified in the requirement, the change in state may not be detected by the API.

ee) The API shall provide an asynchronous notification to alert all application programs when their associated windows change size [41].

ff) The API shall provide a function to allow application programs to reset the display as described in the ATC Controller Standard, Section 7.1.4 [42].

gg) The API shall provide a function to illuminate or extinguish the CPU ACTIVE LED described in the ATC Controller Standard, Section 7 and Section B.2 [43]. The function shall only operate for application programs with a window in focus [44].

hh) The API shall provide a function to send raw output data to the display [45]. If the application window is in focus, the data shall be sent to the display port without interpretation or buffering by the API [46]. If the application window is not in focus, the API shall discard the data [47].

ii) The API shall provide a function to read raw input data from the display (this does not include the Aux Switch which is handled separately; see Item “c”) [48]. This function shall return raw data from the input buffer without the key code interpretation described in item “y” [49].

Table 1. ATC controller escape sequences mapped to API character codes.

Character Pressed on ATC Keypad	Escape Sequence Received By API	Code Returned by API Function to Application Program
Up Arrow	0x1B 0x5B 0x41	0x80
Down Arrow	0x1B 0x5B 0x42	0x81
Right Arrow	0x1B 0x5B 0x43	0x82
Left Arrow	0x1B 0x5B 0x44	0x83
Next	0x1B 0x4F 0x50	0x84
Yes	0x1B 0x4F 0x51	0x85
No	0x1B 0x4F 0x52	0x86
Vendor Specific Key 1	0x1B 0x4F 0x56	0x89
Vendor Specific Key 2	0x1B 0x4F 0x57	0x8A
Vendor Specific Key 3	0x1B 0x4F 0x58	0x8B
Vendor Specific Key 4	0x1B 0x4F 0x59	0x8C
Vendor Specific Key 5	0x1B 0x4F 0x5A	0x8D
Vendor Specific Key 6	0x1B 0x4F 0x5B	0x8E
Vendor Specific Key 7	0x1B 0x4F 0x5C	0x8F
Vendor Specific Key 8	0x1B 0x4F 0x5D	0x90
Vendor Specific Key 9	0x1B 0x4F 0x5E	0x91
Vendor Specific Key 10	0x1B 0x4F 0x5F	0x92
Vendor Specific Key 11	0x1B 0x4F 0x60	0x93
Vendor Specific Key 12	0x1B 0x4F 0x61	0x94
Vendor Specific Key 13	0x1B 0x4F 0x62	0x95
Vendor Specific Key 14	0x1B 0x4F 0x63	0x96
Vendor Specific Key 15	0x1B 0x4F 0x64	0x97
Vendor Specific Key 16	0x1B 0x4F 0x65	0x98

3.1.2 Field I/O Manager Requirements

Historically, a controller unit would have a single application program communicating to one type of Field I/O Device in the cabinet. Application program developers could assume exclusive assignment and control of all of the I/O points of each Field I/O Device. Conversely, the ATC Controller architecture is designed to support multiple application programs and multiple numbers and types of Field I/O Devices simultaneously. Application program developers cannot have exclusive control but must share the I/O points of the Field I/O Device. The API supports this by assigning I/O points of the Field I/O Devices to application programs as they are requested.

Section 8 of the ATC Controller Standard specifies interfaces between the ATC Engine Board and the Field I/O Devices of four transportation cabinet architectures: Model 332

Cabinets, NEMA TS 1 and TS 2 Type 2 Cabinets, NEMA TS 2 Type 1 Cabinets and ITS Cabinets. For those using modules for the ATC 2070 controllers, these correspond to the 2070-2A, 2070-2B/2070-8/MMU, 2070-2N/BIU/MMU and the 2070-2B/SIU/CMU configurations respectively.

The ATC Controller Standard describes a command/response serial message interface between the ATC Engine Board and the Field I/O Devices associated with these architectures. The details of the frames (messages) used by each of the architectures are specified in the ATC Controller Standard except as follows: a) the details for the frames used to communicate with a Serial Interface Unit (SIU) and Cabinet Monitor Unit (CMU) are found in the ITS Cabinet Standard (see Section 1.4) and b) the details for the frames used to communicate with the Bus Interface Unit (BIU) and Malfunction Management Unit (MMU) are found in the NEMA TS 2 Standard (see Section 1.4).

In order to create a common application interface to multiple Field I/O Devices and various types of Field I/O Devices, some parameters or frame types (message types) that are available in “single operational program/single field device type” architectures are not available to application programs through the API. This is to avoid conflicting parameters from multiple application programs. In addition, other frames which are not commonly used are not supported by the API in order to promote simplicity and reliability of the API.

The API does not make any assumptions about the architecture of the cabinet being used by the ATC Controller Unit. Application programs must register with the API, telling the API what port number to use, the Field I/O Device type and Field I/O Device number (there can be multiple numbers of BIUs and SIUs in a single transportation cabinet). The API does not initiate communications to any Field I/O Device unless it is explicitly commanded to do so by an application program.

Figure 8 shows a logical diagram of the Field I/O Manager and its interfaces. The figure shows application programs interfacing to the Field I/O Manager programmatically on the ATC Engine Board. The Field I/O Manager interfaces with Field I/O Devices external to the ATC Engine Board via serial ports. The Field I/O Devices have parallel interfaces to field control devices such as detectors and load switches.

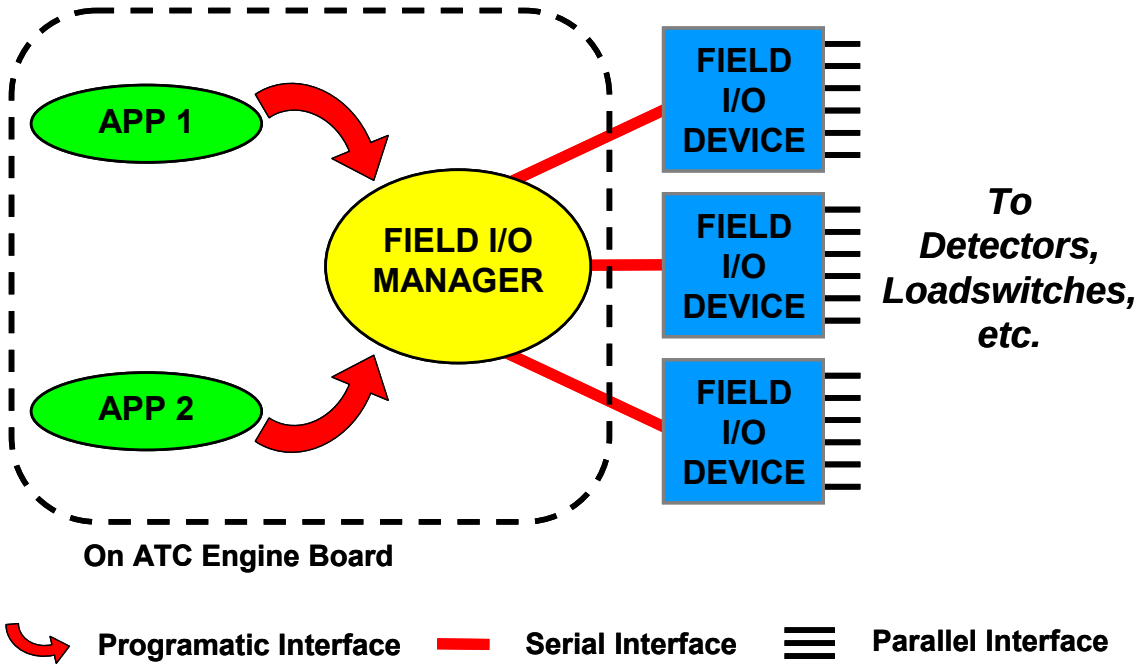

Figure 8. Logical Diagram of the API Field I/O Manager and its Interfaces.

The API requirements for the Field I/O Manager are as follows:

a) The API shall assume it has exclusive access to the serial communications ports of the ATC Engine Board that are designated for Field I/O Devices [1]. The supported Field I/O serial communications ports shall be SP3, SP5 and SP8 [2]. The supported communication modes on those ports shall be 153.6 Kbps and 614.4 Kbps SDLC [3]. The API shall not open any serial communications port or initiate communications to any Field I/O Device unless explicitly commanded to do so by an application program [4].

b) The API shall support all cabinet architectures and associated Field I/O Device types as listed in the ATC Controller Standard Section 8 [5]. The API shall support the Field I/O Device types shown in Table 2 [6]. The API shall assume that BIU and MMU Field I/O Devices operate at 153.6 Kbps and all other Field I/O Device types operate at 614.4 Kbps [7].

   Guidance: Application programs may read serial port and configuration information from the ATC Host Module EEPROM.

Table 2. Field I/O Device types supported by the ATC API.

Cabinet Architecture	Supported Field I/O Device Types	API FIO Device Name
Caltrans Model 332	Parallel Connection to Model 332 Cabinet (2070-2A equivalent device)	FIO332
NEMA TS 1	Parallel Connection to NEMA TS 1 Cabinet (2070-8 equivalent device)	FIOTS1
NEMA TS 2 Type 1	Parallel Connection to NEMA TS 2 Type 1 Cabinet (2070-2N equivalent device), MMU, Detector BIUs, Terminal & Facilities BIUs	FIOTS2, FIOMMU, FIODR1, FIODR2, FIODR3, FIODR4, FIODR5, FIODR6, FIODR7, FIODR8, FIOTF1, FIOTF2, FIOTF3, FIOTF4, FIOTF5, FIOTF6, FIOTF7, FIOTF8
NEMA TS 2 Type 2	Parallel Connection to NEMA TS 2 Type 2 Cabinet (2070-8 equivalent device), MMU	FIOTS1, FIOMMU, FIODR1, FIODR2, FIODR3, FIODR4, FIODR5, FIODR6, FIODR7, FIODR8, FIOTF1, FIOTF2, FIOTF3, FIOTF4, FIOTF5, FIOTF6, FIOTF7, FIOTF8
ITS	CMU, Input SIUs, 6-Pack Output SIUs, 14-Pack Output SIUs	FIOCMU, FIOINSIU1, FIOINSIU2, FIOINSIU3, FIOINSIU4, FIOINSIU5, FIOOUT6SIU1, FIOOUT6SIU2, FIOOUT6SIU3, FIOOUT6SIU4, FIOOUT14SIU1, FIOOUT14SIU2

c) The API shall support communication to multiple Field I/O Devices on a single communications port provided the Field I/O Devices have compatible physical communication attributes [8]. The API shall support a maximum of one Field I/O Device of each type per communications port except in the case of BIUs and SIUs [9]. The API shall support up to 8 Detector BIU and 8 Terminal & Facilities BIU Field I/O Devices per communications port [10]. The API shall support up to 5 Input SIU, 2 14-Pack Output SIU, and 4 6-Pack Output SIU Field I/O Devices per communications port [11]. The API shall only support valid Output SIU combinations as defined in the ITS Cabinet Standard, Section 4.7 [12]. The API shall identify specific Field I/O Devices using the API Field I/O Device Names in Table 2 [13].

Guidance: System integrators need to make sure that they have sufficient processing power and communications bandwidth for the Field I/O Devices they use.

Guidance: The API is NOT required to operate NEMA TS 2 and non TS 2 Field I/O Devices simultaneously on the same communications port.

d) The API shall provide a method for application programs to register and deregister with the API for access to the API Field I/O services [14]. The process of application program registration shall not cause the API to perform any communications with the Field I/O Device [15]. When an application program deregisters for access to Field I/O services, the API shall deregister (as defined in Item "e") all Field I/O devices registered by that application program [16].

e) The API shall provide a method to allow application programs to register and deregister for access to specific Field I/O Devices by specifying the communications port, device type, and where applicable, the Field I/O Device number [17]. Once a device has been registered on a communications port, the API shall permit the registration of additional compatible Field I/O Devices on the same communications port and prohibit the registration of incompatible Field I/O Devices on the same communications port [18]. The Field I/O Device registration process shall not cause the API to perform any device communications [19]. When an application program deregisters for access to a Field I/O Device, the API shall disable (as defined in Item "g") the Field I/O Device, relinquish all output points for that device and set all application program settable states to their default values [20].

f) The API shall provide a method for application programs to query for the presence of a Field I/O Device using the communications port, device type, and where applicable, the Field I/O Device number [21]. If the API does not have the communications port open at the time of the query and it is necessary for the API to open the communications port to determine the Field I/O Device, the API shall close the communications port after the query is completed [22]. If the API has the communications port open at the time of the query and the communications attributes for the Field I/O Device used in the query are not compatible with the current settings on the communications port, the API shall assume that the Field I/O Device is not present [23]. If the API has the communications port open at the time of the query and the API is already successfully completing scheduled communications to the Field I/O Device, the API shall indicate that the Field I/O Device is present without sending any additional frames to the device [24].

Guidance: It is not required for an application program to be registered with a Field I/O Device to perform a query.

g) The API shall provide a method which allows an application program to enable and disable communications to a Field I/O Device for which the

application program has registered [25]. When the communications enable method is called, the API shall initiate scheduled communications between the API and the specified Field I/O Device if not already active [26]. When the disable communications method is called, the API shall cease scheduled communications between the API and the specified Field I/O Device if the device is no longer enabled by any application program [27]. When a Field I/O Device is disabled, any output points which have been reserved by that application program shall be set to Off [28].

h) The API shall provide a method for application programs to read the states of the input and output points on registered Field I/O Devices, including both filtered and non-filtered states for the input points (depending on which input frames are scheduled) [29]. If multiple application programs have registered for the same Field I/O Device, the API shall provide shared read access to the input and output point states for all application programs which have registered that device [30]. When the state of an output point is read, the API shall return the current state of that output point within the API [31].

Guidance: When a read is requested, the API does not attempt to query the actual states of the output points on the Field I/O Device.

i) The API shall provide a method for application programs to reserve/relinquish exclusive “write access” to individual output points of a Field I/O Device [32]. If an application program reserves a point that has already been reserved by that application program, it shall not be considered an error [33]. If an application program relinquishes a point that is already in the relinquished state for that application program, it shall not be considered an error [34]. If a point in a group of points cannot be reserved, the reservation attempt shall fail for all of them [35]. The API shall allow only one application program to reserve write access to any individual output point [36]. The API shall allow multiple application programs to reserve different output points on a single Field I/O Device [37]. Exclusive reservation of an output point for write access by one application program shall not preclude other application programs from reading the state of the output point [38]. The API shall provide error codes so that the application program can determine if the reservation action was successful or if there was a conflict with another application program [39]. The API shall make output point reservations on a “first come first served basis” [40]. An application program shall be able to set the state of an output point if it has registered the associated Field I/O Device and reserved exclusive write access to the output point [41]. To set the state of an output point and control dimming, the API shall use separate arrays for control of the Load Switch + and Load Switch – (see Section 3.3.1.4.1.5 of the TS 2 Standard) [42]. The API shall provide a method for application programs to

query the reservation status of output points on registered Field I/O Devices [43].

Guidance: Non-NEMA devices set their outputs using arrays of Control and Data instead of Load Switch +/-. The API needs to make the translation accordingly.

j) The API shall provide a method for application programs to map/unmap reserved output points to reserved channels and colors on a registered FIOMMU or FIOCMU device [44]. The API shall use this mapping to set the contents of FIOMMU Frame 0 and FIOCMU Frames 61 and 67 [45]. Any channel and color not mapped to an output point shall be set to Off [46]. The API shall provide a method for application programs to reserve/relinquish exclusive control of individual monitored channels on the FIOMMU or FIOCMU device [47]. If an application program reserves a channel that has already been reserved by that application program, it shall not be considered an error [48]. If an application program relinquishes a channel that is already in the relinquished state for that application program, it shall not be considered an error [49]. If a channel in a group of channels cannot be reserved, the reservation attempt shall fail for all of them [50]. The API shall allow multiple applications to reserve different channels on a single FIOMMU or FIOCMU device [51]. The API shall provide error codes so that the application program can determine if the reservation action was successful or if there was a conflict with another application [52]. The API shall make channel reservations on a “first come first served basis” [53]. The API shall provide a method for applications to query the reservation status of channels on registered FIOMMU or FIOCMU devices [54]. Relinquishing a reserved output point or channel shall clear the associated assignments [55].

k) The API shall provide functions which allow application programs to set and get the leading and trailing edge filter values on a per input basis for all Field I/O Devices that support configurable filtered inputs [56]. If multiple application programs set the filter values of an input, the shortest filter values shall be used [57]. The API shall provide a return code containing the status and the value used for the set filter operation [58]. The default leading and trailing edge filter values shall be 5 consecutive samples [59].

Guidance: Any application program can read and set the filter value on any input. Future versions of the API may want to restrict this in some fashion.

l) The API shall have the ability to collect and buffer the transition buffer information for each registered Field I/O Device used for input [60]. When the API reads the transition buffer of a Field I/O Device, it shall read the entire transition buffer [61]. The API shall buffer the transition data on a per

application program basis with the capability of storing 1024 transition entries in a FIFO fashion [62]. The API shall provide a function which allows application programs to enable or disable transition monitoring of selected input points [63]. By default, transition monitoring for all input points shall be disabled [64]. If an application program enables an input point for transition monitoring and that input point is already in the enabled state, it shall not be considered an error [65]. If an application program disables an input point for transition monitoring and that input point is already in the disabled state, it shall not be considered an error [66]. The API shall provide functions that allow application programs to access the API transition buffer information asynchronously (i.e. read the transition entries from the API buffer independent of any Field I/O Device communications) [67]. When an application program reads a transition entry from an API transition buffer, that transition entry shall be cleared for that application program only, without affecting the API transition buffers for other application programs [68]. If the transition buffer in the Field I/O Device overruns before information can be copied to the API transition buffer information, the API shall indicate that a device overrun condition has occurred in the transition buffer for that Field I/O Device [69]. If the transition buffer of the API overruns before the information is retrieved by the application program, the API shall indicate that an API overrun condition has occurred [70].

Guidance: The purpose of the two overrun conditions discussed in the preceding paragraph is to alert application programs that they are missing data.

m) The ATC Controller Standard, Section 8, specifies the frames for communication with Field I/O Devices for Model 332 Cabinets, NEMA TS 1 and TS 2 Type 2 Cabinets and ITS Cabinets. The API shall support a subset of these frames at the scheduled frame frequencies as shown in Table 3 [71]. The NEMA TS 2 Standard, Section 3.3, specifies the frames for communication with Field I/O Devices for NEMA TS 2 Type 1 Cabinets. The API shall support a subset of these frames at the scheduled frame frequencies as shown in Table 4 [72]. The timing for the command/response cycle of the frames shall be defined by the “Handshaking” algorithm in Section 3.3.1.5.3 of the NEMA TS 2 Standard [73].

Guidance: Although the timing for the command/response cycle of the frames is defined for the API by the NEMA TS 2 Standard, it applies to all frame types regardless of the cabinet standard.

n) The API shall provide a method for application programs to set/get the scheduled frame frequencies for a registered Field I/O Device [74]. The frame frequency used by the API shall be the highest frequency requested by all

application programs registered for that Field I/O Device [75]. The API shall provide a method to send a frame from either Table 3 or Table 4 one time (non-scheduled) [76].

Guidance: A frame intended to be sent one time should be sent after the next set of scheduled frames has complete. Generally, the API should order the frames sent to a Field I/O Device based on their frame type.

Guidance: If an application program deregisters with a Field I/O Device, the frame frequency for all frames associated with that application program is set to 0 and the next highest frequencies for those frames are used by the API.

Guidance: For a frame to be disabled to a Field I/O Device, all registered application programs must set the frequency of the frame to 0.

o) The API provides methods to set and clear application-initiated "fault" conditions. The management of the fault condition varies by Field I/O Device type as follows:

i) The API shall provide a method to set/get the Failed State Action of a FIOCMU Field I/O Device [77]. The Failed State Action shall be settable to None (LFSA=0, NFSA=0), Latched (LFSA=1, NFSA=0), or Non Latched (LFSA=0, NFSA=1) [78]. The default Failed State Action shall be None [79]. If any application program sets the state to Latched, the API shall set the Failed State Action to Latched [80]. If no application program has set the Failed State Action to Latched, then if any application program sets the state to Non Latched, the API shall set the Failed State Action to Non Latched [81]. If all application programs have a state of None, then the API shall set the Failed State Action to None [82].

ii) The API shall provide a method to set/get the state of the Fault Monitor output point of FIOTS1 and FIOTS2 Field I/O Devices [83]. The API shall retain ownership of the Fault Monitor output point and not allow application programs to reserve this output point [84]. If any application program sets the Fault Monitor state to Off, the API shall turn Off the Fault Monitor output point on that device [85]. If all application programs have a Fault Monitor state of On for a FIOTS1 or FIOTS2 Device, then the API shall turn On the Fault Monitor output point on that device [86]. The default state of the Fault Monitor output point shall be On [87].

iii) The API shall provide a method to set/get the state of the Voltage Monitor output point of a FIOTS1 Field I/O Device [88]. The API shall

retain ownership of the Voltage Monitor output point and not allow application programs to reserve this output point [89]. If any application program sets the Voltage Monitor state to Off, the API shall turn Off the Voltage Monitor output point on that device [90]. If all application programs have a Voltage Monitor state of On for a FIOTS1 Device, then the API shall turn On the Voltage Monitor output point on that device [91]. The default state of the Voltage Monitor output point shall be On [92].

iv) The API shall provide a method which allows application programs to assign the output point used for the Watchdog output of any registered Field I/O Device [93]. The API shall restrict the ability to assign the Watchdog output point to the first application program to call the assignment method [94]. The API shall retain ownership of the Watchdog output point and not allow application programs to reserve that output point directly [95].

The API shall provide a method for application programs to register for shared control of the Watchdog output point [96]. The API shall provide a method for Watchdog registered application programs to "request" that the API toggle the state of the Watchdog output point [97]. The API shall only toggle the Watchdog output point if all Watchdog registered application programs have made the toggle request (Watchdog Triggered Condition) [98]. Upon a Watchdog Triggered Condition, the API shall toggle the state of the Watchdog output point within the API [99]. When the API updates the output states of the Field I/O Device (see Item "n"), the API shall clear all previous toggle requests and the Watchdog Triggered Condition so that a new Watchdog Triggered Condition can be generated [100]. The API shall not toggle the Watchdog output point more than once per update of the output states on the Field I/O Device [101].

p) The API shall provide functions which allow application programs to obtain status information of a registered Field I/O Device [102]. All counters contained in the Field I/O Device status information shall be four byte unsigned values each with a maximum value of 4,294,967,295 [103]. The counters shall be frozen when they reach the maximum value to prevent rollover [104]. The API shall provide the following communication status information for each registered Field I/O Device:
   i) Communications Enabled/Disabled;
   ii) Cumulative successful response count for all frames to this device;
   iii) Cumulative error count for all frames to this device; and
   iv) Command frames sent to this device with the following information for each frame type: current scheduled frequency, cumulative successful

response count, cumulative error count, numbers of errors in the last 10 frames, a response frame sequence number, frame size in bytes and the raw data from the most recent response frame [105]. The response frame sequence number shall be a four byte unsigned value and rollover after the maximum value [106].

The API shall provide a method for application programs to reset the communications status counters to 0 (zero) for a registered Field I/O Device [107]. A response frame shall only be considered successful if it is fully received within the time period defined by the "Handshaking" algorithm in Section 3.3.1.5.3 of the NEMA TS 2 Standard [108].

Guidance: Although the response frame time period is defined for the API by the NEMA TS 2 Standard, it applies to all frame types regardless of the cabinet standard.

q) The API shall provide an API Health Monitor Function which registered application programs use to indicate to the API that they are operational [109]. The API shall provide a method to set an API Health Monitor Timeout for each application program (each application program has its own unique API Health Monitor Timeout) [110]. This API Health Monitor Timeout shall indicate the maximum allowable time between calls to the API Health Monitor Function [111]. The API Health Monitor Timeout shall be specified in tenths of a second [112]. If the API Health Monitor Timeout expires for an application program, the API shall disable (as defined previously in Item "g") all Field I/O Devices registered by that application program [113]. The API shall provide a method for an application program to disable the API Health Monitor feature for itself [114]. The API shall provide a method for an application program to reset an API Health Monitor fault condition and allow the API to resume Field I/O Device communications [115]. An application program shall only be able to reset its own API Health Monitor fault condition and not that of any other application program [116]. If an application program resets the API Health Monitor fault condition, then any devices that were disabled due to that condition shall be re-enabled [117]. If an application program attempts to enable a device (as defined in Item "g") that has been disabled due to an API Health Monitor fault condition, then the enable operation shall return an error and the Field I/O Device remain disabled [118]. A call to the API Health Monitor Function after a Health Monitor fault has occurred shall not reset the Health Monitor fault condition [119]. The API Health Monitor Function shall return whether an API Health Monitor fault condition exists [120].

r) The API shall provide a method for an application program to send the Get CMU Configuration frame to a registered FIOCMU device [121].

s) The API shall reset all Module Status bits using the Request Module Status frame when a FIO332, FIOTS1, FIOTS2 or SIU device is first Enabled (as defined in Item “g”) [122]. Anytime a response to a Request Module Status frame has Module Status bits indicating hardware reset, comm loss, or watchdog reset, then the API shall clear those bits, reset the input point filter values (Item “k”) and reconfigure transition reporting (Item “l”) [123].

t) The API shall provide a method to notify an application program when a command frame is acknowledged (response frame received by the API) or when an error occurs [124]. The command frame shall be identified by the frame type and a registered Field I/O Device [125]. The response frame notification shall include the Field I/O Device, response frame type, response frame sequence number, response frame size in bytes and an indication as to why the notification occurred (response received or error detected) [126]. The notification shall be able to be set for a one time occurrence or continuous occurrence [127].

u) The API shall provide a method to set and get the Dark Channel Map selection for a registered FIOCMU device [128]. If multiple application programs attempt to set the Dark Channel Map selection, the API shall use the most recent selection [129]. The default value of the Dark Channel Map Select bits shall be 0 (Mask #1) [130].

v) The API shall provide a method to set and get the state of the Load Switch Flash bit of a registered FIOMMU device [131]. If multiple application programs attempt to set the state of the Load Switch Flash bit, the API shall use the most recent state [132]. The default value of the Load Switch Flash bit shall be 0 [133].

w) When an application program exits or terminates for any reason, the API shall deregister the application program from the API (as defined in Item “d”) [134].

Table 3. Frame types supported by the API for Model 332 Cabinets, NEMA TS 1 and TS 2 Type 2 Cabinets and ITS Cabinets.

Cmd Frame	Rsp Frame	Description	Frequency & Use
49	177	Request Module Status	Configurable schedule*, default 10 Hz.
51	179	Configure Inputs	Application program driven. Not scheduled.
52	180	Poll Raw Input Data	Configurable schedule*, default 0 Hz (frame not sent).
53	181	Poll Filtered Input Data	Configurable schedule*, default 10 Hz.
54	182	Poll Input Transition Buffer	Configurable schedule*, default 0 Hz (frame not sent).
55	183	Command Outputs	Configurable schedule*, default 10 Hz.
60	188	I/O Module Identification	Application program driven. Not scheduled.
61		Switch Pack Drivers	Configurable schedule*, default 0 Hz (frame not sent).
	189	CMU Status	Response per ITS Cabinet Standard, Section 4.14.16.
62		Send to Local Flash Command	Application program driven. Not scheduled.
	190	Send to Local Flash Response	Response per ITS Cabinet Standard, Section 4.14.16.
65		Get CMU Configuration	Application program driven. Not scheduled.
	193	CMU Configuration	Response per ITS Cabinet Standard, Section 4.14.16.
66		Time and Date Command	1 Hz. Used as defined in the ITS Cabinet Standard, Section 4.14.16.
67		Switch Pack Drivers	Configurable schedule*, default 10 Hz.
	195	CMU Short Status	Response per ITS Cabinet Standard, Section 4.14.16.

* The following configurable scheduled frame frequencies are supported: 0, 1, 2, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100 Hz.

Table 4. Frame types supported by the API for NEMA TS 2 Type 1 Cabinets.

Cmd Frame	Rsp Frame	Description	Frequency & Use
0		Load Switch Drivers	10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	128	MMU (Type 0 ACK)	Response per NEMA TS 2 Standard, Section 3.3.
1		MMU Inputs/Status Request	10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	129	MMU Inputs/Status (Type 1 ACK)	Response per NEMA TS 2 Standard, Section 3.3.
3		MMU Programming Request	1 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	131	MMU Programming (Type 3 ACK)	Response per NEMA TS 2 Standard, Section 3.3.
9		Date and Time Broadcast to All	1 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
10		TF BIU #1 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	138	TF BIU #1 Inputs (Type 10 ACK)	Response per NEMA TS 2 Standard, Section 3.3
11		TF BIU #2 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	139	TF BIU #2 Inputs (Type 11 ACK)	Response per NEMA TS 2 Standard, Section 3.3
12		TF BIU #3 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	140	TF BIU #3 Inputs (Type 12 ACK)	Response per NEMA TS 2 Standard, Section 3.3
13		TF BIU #4 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3
	141	TF BIU #4 Inputs (Type 13 ACK)	Response per NEMA TS 2 Standard, Section 3.3
18		Output Transfer Frame Broadcast to TF BIUs	10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
20		DR BIU #1 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	148	DR BIU #1 Inputs (Type 20 ACK)	Response per NEMA TS 2 Standard, Section 3.3
21		DR BIU #2 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.

Cmd Frame	Rsp Frame	Description	Frequency & Use
	149	DR BIU #2 Inputs (Type 21 ACK)	Response per NEMA TS 2 Standard, Section 3.3
22		DR BIU #3 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	150	DR BIU #3 Inputs (Type 22 ACK)	Response per NEMA TS 2 Standard, Section 3.3
23		DR BIU #4 Outputs/Inputs Request	Configurable schedule*, default 10 Hz. Used as defined in the NEMA TS 2 Standard, Section 3.3.
	151	DR BIU #4 Inputs (Type 23 ACK)	Response per NEMA TS 2 Standard, Section 3.3

* The following configurable scheduled frame frequencies are supported: 0, 1, 2, 5, 10, 20, 30, 40, 50, 60, 70, 80, 90 and 100 Hz.

3.2 API Utility Requirements

The API shall provide a method to determine the version number(s) of the API [1]. The API shall provide a function to allow application programs to set the system time [2].

3.2.1 ATC Configuration Window Requirements

The API shall provide a window called the ATC Configuration Window [1]. Operational Users shall be able to view ATC configuration information on this window provided by the Linux *uname()* function (ATC Controller Standard, Section 2.2.5), the API's version number(s) and the ATC Host Module EEPROM information (ATC Controller Standard, Annex B) [2]. The default ATC Configuration Window size shall be 8 lines x 40 characters with the format as show in Figure 9 [3].

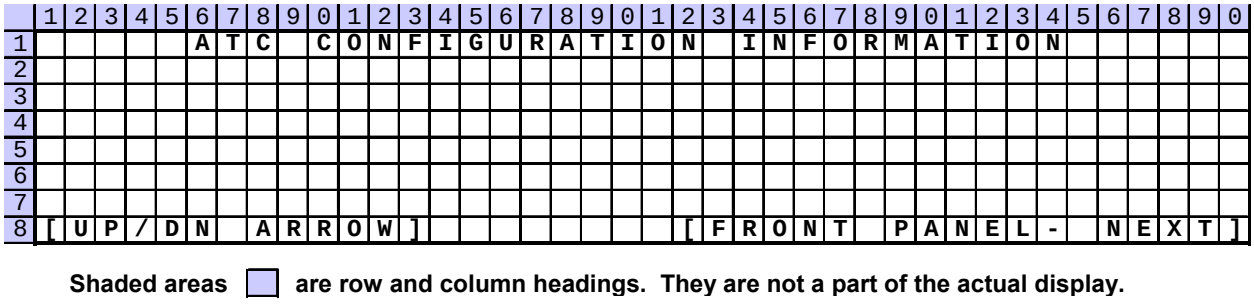

Figure 9. The ATC Configuration Window.

a) The top two lines and bottom line of the ATC Configuration Window shall be fixed as shown in Figure 8 [4].

b) The number of lines between the second line and bottom lines used for displaying the ATC configuration information shall vary according to the size of the ATC display [5].

c) The Operational User shall be able to scroll up and down the ATC Configuration Window one line at a time to view the configuration information using the up and down arrow keys of the controller keypad [6].

d) The Operational User shall be able to put the Front Panel Manager in focus by pressing {<NEXT>} in the ATC Configuration Window [7].

3.3 Performance Requirements

Performance of the API is largely dependent on the efficiency of the ATC BSP. Section 3.1.2 identifies performance requirements for the message interface to Field I/O devices. Otherwise, there are no general performance requirements for the API.

3.4 Design Constraints

The API shall operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard [1]. The API function calls shall be specified using the C programming language as described by “ISO/IEC 9899:1999,” commonly referred to as the C99 Standard [2].

3.5 Software System Attributes

This section identifies requirements that have to do with overall system attributes of the API software.

3.5.1 Portability

It is not required that the implementation of this standard is portable. The API specified by this standard facilitates portability of application programs by defining a common software interface for application programs operating on ATC controller units.

3.5.2 Consistency

The operational look and feel of user interfaces developed for the API shall have consistent window titling conventions, scrolling methods, menu styles and selection methods [1]. If API functions have a similar operation to existing Linux functions, they shall have a similar name and argument style to those functions to the extent possible without causing compilation issues [2]. The API function names shall be lower case [3]. API functions shall use the Linux “errno” error notification mechanism if an error indication is expected for a function [4]. The API shall be loadable as an ELF (Executable and Linking Format) library [5].

3.6 Other Requirements

There are no other requirements for this standard.

4 APPLICATION PROGRAMMING INTERFACE

This section specifies the ATC API as function calls using the C programming language as described by “ISO/IEC 9899:1999” commonly referred to as the C99 Standard. The ATC API function calls are divided into three subsections: “Front Panel Manager Functions,” “Field I/O Manager Functions” and “Utility Functions.” These subsections correspond to the manager and utility requirement subsections of similar names in Section 3. Within the subsections, each API function is specified using a Linux online documentation format called “man page” format. The man pages are written with the assumption that the reader is familiar with C and the Linux operating system. The fields of the man page are described below.

- NAME. This field has a standardized format consisting of the function name, followed by a dash, followed by a short (usually one line) description of the functionality that is to be provided.

- SYNOPSIS. This field gives a short overview on available function options including the corresponding header files and the function prototype. This provides the programmer with the number and type of arguments to the function and function’s return type.

- DESCRIPTION. This field contains detailed information about the function. It explains how the function is used, what the arguments are for, algorithms used (if any) and file formats (if any).

- RETURN VALUES. This field describes values returned by the function that are not the return value of the function.

- ERRORS. This field lists any error codes that might be returned and the possible cause. If a library is implemented using a Linux kernel driver, standard Linux system call error codes may be returned in addition to those listed.

- NOTES. This field contains any other function information that a programmer might need to know.

- RESTRICTIONS. This field contains any restrictions imposed on or by the function’s operation.

- SEE ALSO. This field provides a list of related man pages listed in alphabetical order.

When function names are used within the man page (other than the NAME and SYNOPSIS fields), they have the form “functionname(*manpagesection*)” where

"*manpagesection*" refers to one of the 9 sections of the Linux man pages. In this document *manpagesection* may have the following values:

"2" Indicates the function is a Linux system call (Ex. open(2), close(2));

"3" Indicates the function is a part of the standard C library (Ex. fopen(3), fclose(3));

"3fpui" Indicates the function is a part of the API Front Panel User Interface library (Ex. fpui_open(3fpui), fpui_close(3fpui));

"3fio" Indicates the function is a part of the API Field I/O library (Ex. fio_register(3fio), fio_deregister(3fio)); and

"3tod" Indicates the function is a part of the API Time of Day library (Ex. tod_set(3tod), tod_get_timesrc(3tod)).

For readability purposes, each man page starts at the top of a new page.

4.1 Front Panel Manager Functions

This section specifies the API Front Panel Manager functions. They are listed alphabetically with each specification beginning on a new page. Operationally, they can be grouped as shown below.

- General Functions – fpui_apiver, fpui_open, fpui_close, fpui_get_window_size, fpui_get_focus, fpui_clear, fpui_refresh and fpui_set_emergency.
- Attribute Functions – fpui_set_window_attr, fpui_get_window_attr, fpui_set_character_blink, fpui_get_character_blink, fpui_set_backlight, fpui_get_backlight, fpui_set_backlight_timeout, fpui_set_cursor_blink, fpui_get_cursor_blink, fpui_set_reverse_video, fpui_get_reverse_video, fpui_set_underline, fpui_get_underline, fpui_set_auto_wrap, fpui_get_auto_wrap, fpui_set_auto_repeat, fpui_get_auto_repeat, fpui_set_cursor, fpui_get_cursor, fpui_set_auto_scroll, fpui_get_auto_scroll and fpui_reset_all_attributes.
- Read Functions – fpui_poll, fpui_read, fpui_read_char and fpui_read_string.
- Write Functions – fpui_write, fpui_write_char, fpui_write_string, fpui_write_at, fpui_write_char_at and fpui_write_string_at.
- Cursor Functions – fpui_get_cursor_pos, fpui_set_cursor_pos, fpui_home, fpui_set_tab and fpui_clear_tab.
- Special Character Functions – fpui_compose_special_char and fpui_display_special_char.
- LED Functions – fpui_set_led and fpui_get_led.
- Aux Switch Functions – fpui_open_aux_switch, fpui_close_aux_switch and

fpui_read_aux_switch.

- Key Mapping Functions – fpui_set_keymap, fpui_get_keymap, fpui_del_keymap and fpui_reset_keymap.

The header file, “fpui.h,” defines all the function calls in a library named “libfpui.so.”

NAME

fpui_apiver – Obtain version information about the FIO API

SYNOPSIS

#include <fpui.h>

char * fpui_apiver(fpui_handle *hdl*, **int** *type* **)**

DESCRIPTION

The **fpui_apiver**(3fpui) library call will return a string containing the version number of either the API itself or the underlying driver as requested.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

type is 1 for the API version ID and 2 for the driver version ID.

RETURN VALUE

fpui_apiver(3fpui) will return a valid char * containing the requested version ID. On error, NULL will be returned with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

When requesting the API version ID, the *hdl* parameter is not used. When requesting the driver version ID, a valid *hdl* parameter is required.

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_clear – Clear screen

SYNOPSIS

#include <fpui.h>

int fpui_clear(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_clear**(3fpui) library call clears the screen without moving the cursor.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors

NOTES

This routine may be implemented by wrapping an **fpui_write_string**(3fpui) call. It sends the string "<ESC>[2J" (0x1b 0x5b 0x32 0x4a).

RESTRICTIONS

If the window is not in focus, the window will still be cleared.

SEE ALSO

fpui_write_string(3fpui)

NAME

fpui_clear_tab – Clear tab stop

SYNOPSIS

#include <fpui.h>

int fpui_clear_tab(fpui_handle *hdl* , **int** *type* **)**

DESCRIPTION

The **fpui_clear_tab**(3fpui) library call will clear the stop tab at the cursor position when *type* is 0, 1, or 2. It will clear all stop tabs when *type* is 3.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).
type is 0, 1, or 2 to clear the stop tab at the cursor position. Or, 3 to clear all stop tabs.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

EINVAL *type* is invalid.
Refer to **fpui_write_string**(3fpui) for additional errors

NOTES

This routine may be implemented by wrapping an **fpui_write_string**(3fpui) call. It sends the string "<ESC>[*Pn*g" (0x1b 0x5b *Pn* 0x67) where *Pn* is the *type* of tab to clear.

RESTRICTIONS

None

SEE ALSO

fpui_set_tab(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_close – Close a terminal interface

SYNOPSIS

#include <fpui.h>

int fpui_close(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_close**(3fpui) closes a terminal interface and its descriptor. Any resources allocated to the terminal are returned to the system for reuse.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

EBADF	*hdl* isn't a valid open descriptor.
EINTR	The **fpui_close**(3fpui) call was interrupted by a signal.
EIO	An I/O error occurred.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_open(3fpui)

NAME

fpui_close_aux_switch – Close the Aux Switch interface

SYNOPSIS

```
#include      <fpui.h>

int fpui_close_aux_switch( )
```

DESCRIPTION

The **fpui_close_aux_switch**(3fpui) releases exclusive access and closes the Aux Switch interface. Any resources allocated when opened are returned to the system for reuse.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

EBADF	The underlying object or device used to access the Aux Swith is invalid.
EINTR	The **fpui_close_aux_switch**(3fpui) call was interrupted by a signal.
EIO	An I/O error occurred.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_open_aux_switch(3fpui), **fpui_read_aux_switch**(3fpui)

NAME

fpui_compose_special_char – Define a special bit mapped character

SYNOPSIS

#include <fpui.h>

int fpui_compose_special_char(fpui_handle *hdl*, **int** *index*, **unsigned char *** *buf*)

DESCRIPTION

The **fpui_compose_special_char**(3fpui) library call will allow the application to define a special bit mapped character which can be referenced by *index*. Each special character is 8 pixels by 8 pixels and encoded in the referenced 8 byte array *buf*.

Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
index	is the index of the character to be composed.
buf	is the reference to an 8 byte array containing the pixels.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>P*P1*[*Pn*;*Pn* .. f" (0x1b 0x50 *P1* 0x5b *Pn* 3b *Pn* .. 0x66)

Where *P1* is the *index* and *Pn* are the 8 ASCII encoded decimal bytes that form the character bit map.

RESTRICTIONS

Only 8 special characters can be defined.

SEE ALSO

fpui_display_special_char(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_del_keymap – Delete keymap entry

SYNOPSIS

#include <fpui.h>

int fpui_del_keymap(fpui_handle *hdl*, **char** *key* **)**

DESCRIPTION

The **fpui_del_keymap**(3fpui) library call is used to delete the first keymap entry in which the key value matches the specified key value. If no matching entry is found, no action is taken.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).
key is the single byte code used to locate the keymap entry.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui)

NOTES

There are no checks for duplicate *key* values or duplicate escape sequences. Escape sequences are compared in the order they appear in the list. New *key* mappings are inserted in the first available slot in the list, which may be a slot vacated by an earlier delete operation. This means that the list may not be in the order mappings were added. **fpui_del_keymap**(3fpui) will remove the first occurrence of the specified *key* from the list. A second, duplicate entry would require another **fpui_del_keymap**(3fpui) operation.

RESTRICTIONS

None

SEE ALSO

fpui_set_keymap(3fpui), **fpui_get_keymap**(3fpui), **fpui_reset_keymap**(3fpui)

NAME

fpui_display_special_char – Define a special bit mapped character

SYNOPSIS

#include <fpui.h>

int fpui_display_special_char(fpui_handle *hdl*, **int** *index* **)**

DESCRIPTION

The **fpui_display_special_char**(3fpui) library call will display a previously composed character at the current cursor location.

Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
index	is the index of the character to be displayed.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[<*Pn*V" (0x1b 0x5b 0x3c *Pn* 0x56).

Where *Pn* is the index of the composed character.

RESTRICTIONS

Only 8 special characters can be defined.

SEE ALSO

fpui_compose_special_char(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_get_auto_repeat – Get the auto repeat state

SYNOPSIS

#include <fpui.h>

boolean fpui_get_auto_repeat(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_auto_repeat**(3fpui) library call will return the boolean state of auto repeat mode. When auto repeat is enabled, all key strokes from the front panel will repeat as long as the key is help depressed. When this mode is not enabled there will be only one key code sent per key press.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_auto_repeat(3fpui) will return **FALSE** if auto repeat mode is off, **TRUE** if the mode is on, and -1 if an error occurred with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This may be implemented by wrapping an **fpui_get_attributes**(3fpui) call.

RESTRICTIONS

None

SEE ALSO

fpui_get_attributes(3fpui), **fpui_set_auto_scroll**(3fpui), **fpui_set_auto_wrap**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_get_auto_scroll – Get the auto scroll state

SYNOPSIS

#include <fpui.h>

boolean fpui_get_auto_scroll(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_auto_scroll**(3fpui) library call will return the boolean state of auto scroll mode. Auto scroll mode controls the display response when the cursor is on the bottom most line of the display and a new line is sent. If auto scroll is **TRUE**, the display will scroll all lines on the display up one line, overwriting the top most line. If auto scroll is **FALSE,** the display remains intact but the cursor will move to the top most line. In either case the column position of the cursor is not effected by the new line.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_auto_scroll(3fpui) will return **FALSE** if auto scroll mode is off, **TRUE** if the mode is on and -1 if an error occurred with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This may be implemented by wrapping an **fpui_get_attributes**(3fpui) call.

RESTRICTIONS

None

SEE ALSO

fpui_get_attributes(3fpui), **fpui_set_auto_repeat**(3fpui), **fpui_auto_wrap**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_get_auto_wrap – Get the auto wrap state

SYNOPSIS

#include <fpui.h>

boolean fpui_get_auto_wrap(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_auto_wrap**(3fpui) library call will return the boolean state of auto wrap mode. Auto wrap mode controls the displays response when the cursor is in the right most column and a character is sent. If auto wrap mode is **TRUE** the character is displayed at the current cursor position and the cursor is wrapped back to the beginning of the line. If auto wrap mode is **FALSE** the character is displayed at the current cursor position and the cursor will remain in the right most column. Any subsequent characters will overwrite the previous ones.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_auto_wrap(3fpui) will return **FALSE** if auto wrap mode is off, **TRUE** if the mode is on and -1 if an error occurred with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This may be implemented by wrapping an **fpui_get_attributes**(3fpui) call.

RESTRICTIONS

None

SEE ALSO

fpui_get_attributes(3fpui), **fpui_set_auto_repeat**(3fpui), **fpui_set_auto_scroll**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_get_backlight – Get the current state of the Backlight

SYNOPSIS

#include <fpui.h>

boolean fpui_get_backlight(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_backlight**(3fpui) library call will return the boolean state of the backlight.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_backlight(3fpui) will return **FALSE** if the backlight is off, **TRUE** if the backlight is on and -1 if an error occurred with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This may be implemented by wrapping an **fpui_get_attributes**(3fpui) call.

RESTRICTIONS

None

SEE ALSO

fpui_set_backlight(3fpui), **fpui_get_attributes**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_get_character_blink – Get the current character blink state

SYNOPSIS

#include <fpui.h>

boolean fpui_get_character_blink(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_character_blink**(3fpui) library call will return the boolean state of character blink mode of the character at the current cursor position. Since the display cannot return this information directly, the attribute is maintained by the window itself.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_character_blink(3fpui) will return **FALSE** if character blink mode is off, **TRUE** if the mode is on and -1 if an error occurred with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

None

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_get_cursor – Get the cursor state

SYNOPSIS

#include <fpui.h>

boolean fpui_get_cursor(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_cursor**(3fpui) library call will return the boolean state on the cursor. Since the display cannot return this information directly, the attribute is maintained by the window itself.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_cursor(3fpui) will return **FALSE** if the cursor is off, **TRUE** if the cursor is on and -1 if an error occurred with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_set_cursor(3fpui)

NAME

fpui_get_cursor_blink – Get the cursor blink state

SYNOPSIS

#include <fpui.h>

boolean fpui_get_cursor_blink(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_cursor_blink**(3fpui) library call will return the boolean state on the cursor blink. Since the display cannot return this information directly, the attribute is maintained by the window itself.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_cursor_blink(3fpui) will return **FALSE** if the cursor is not set to blink, **TRUE** if the cursor is set to blink and -1 if an error occurred, with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

None

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_get_cursor_pos – Get the current Cursor Position

SYNOPSIS

#include <fpui.h>

int fpui_get_cursor_pos(fpui_handle *hdl*, **int *** *row*, **int *** *column* **)**

DESCRIPTION

The **fpui_get_cursor_pos**(3fpui) library call returns the current position of the cursor in the *row* and *column* arguments.

Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
row	is the reference of an integer storage location.
column	is the reference of an integer storage location.

RETURN VALUE

On success, the current cursor position is loaded into the locations referenced by *row* and *column* and 0 (zero) is returned. On error, the *row* and *column* variables remain unchanged and -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[6n" (0x1b 0x5b 0x36 0x60).

It expects a response of the form "<ESC>[*Py*;*Px*R" (0x1b 0x5b *Py* 0x3b *Px* 0x52) where *Py* and *Px* are the *row* and *column* indexes.

RESTRICTIONS

None

SEE ALSO

fpui_set_cursor_pos(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_get_focus – Get the current focus state

SYNOPSIS

#include <fpui.h>

boolean fpui_get_focus(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_focus**(3fpui) library call will return **TRUE** if the calling application currently has focus and **FALSE** if it does not. Since the display cannot return this information directly, the attribute is maintained by the window itself.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

On success, **TRUE** will be returned if the calling application currently has focus and **FALSE** if it does not. On error, -1 is returned with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

There is no function for an application to grant its window to be in focus.

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_get_keymap – Get a keymap entry

SYNOPSIS

#include <fpui.h>

int fpui_get_keymap(fpui_handle *hdl*, **int** *key*, **char *** *str*, **int** *size* **)**

DESCRIPTION

The **fpui_get_keymap**(3fpui) operation will return, in the argument *str*, the escape sequence associated with the value of *key*. If an entry corresponding to the value of *key* is not found, the first character in the *str* array will be **NULL**. The argument *str* will be **NULL** terminated if the size of *str* is greater than the size of the escape sequence plus 1 (for the **NULL** character). Otherwise the escape sequence will be packed to the size of the *str* array and left unterminated.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).
key is the single byte code used to locate the keymap entry.
str a character array to return the escape sequence in.
size the size, in characters, of the string array *str*.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui)

NOTES

There are no checks for duplicate *key* values or escape sequences. *Key* values are compared in the order they appear in the list. New *key* mappings are inserted in the first available slot in the list, which may be a slot vacated by an earlier delete operation. This means that the list may not be in the order mappings were added. **fpui_get_keymap**(3fpui) will return the first occurrence of the specified *key* from the list. A second, duplicate entry would not be accessible until the previous entry has been deleted.

RESTRICTIONS

None

SEE ALSO

fpui_set_keymap(3fpui), **fpui_del_keymap**(3fpui), **fpui_reset_keymap**(3fpui)

NAME

fpui_get_led – Get the current state of the status LED

SYNOPSIS

#include <fpui.h>

boolean fpui_get_led(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_led**(3fpui) library call will return the boolean state of the status LED. Each window maintains its own state of the LED. The window handles updating and controlling the actual Front Panel LED when the application gains focus so this call will always return the window's LED state.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_led(3fpui) will return **FALSE** if the LED is off, **TRUE** if the LED is on and -1 if an error occurred with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

None

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_get_reverse_video – Get the state of reverse video mode

SYNOPSIS

#include <fpui.h>

boolean fpui_get_reverse_video(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_reverse_video**(3fpui) library call will return the boolean state of reverse video mode for the character at the current cursor position.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_reverse_video(3fpui) will return **TRUE** if reverse video mode is currently active on the character at the current cursor position, and **FALSE** if reverse video mode is not active. On error, -1 will be returned with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

The Front Panel hardware does not support this operation directly. The inquiry is serviced by the window which maintains this information for every character on the screen.

RESTRICTIONS

None

SEE ALSO

fpui_set_reverse_video(3fpui)

NAME

fpui_get_underline – Get the state of underline mode

SYNOPSIS

#include <fpui.h>

boolean fpui_get_underline(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_underline**(3fpui) library call will return the boolean state of underline mode for the character at the current cursor position.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

fpui_get_underline(3fpui) will return **TRUE** if underline mode is currently active on the character at the current cursor position and **FALSE** if underline mode is not active. On error, -1 will be returned with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

The Front Panel hardware does not support this operation directly. The inquiry is serviced by the window which maintains this information for every character on the screen.

RESTRICTIONS

None

SEE ALSO

fpui_set_underline(3fpui)

NAME

fpui_get_window_attr – Get the current Window attributes

SYNOPSIS

#include <fpui.h>

int fpui_get_window_attr(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_get_window_attr**(3fpui) library call will return a bit field encoding of the current window attributes. The bitfields are encoded as follows:

```
union {
        int             errcode;
        unsigned int    auto_wrap:1,
                        auto_scroll:1,
                        auto_repeat:1,
                        backlight:1,
                        :4,
                        backlight_timeout:8,
                        :16;
}
```

Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

On success, **fpui_get_window_attr**(3fpui) will return the current window attributes encoded as discussed above. On error, the field *errcode* will be -1 with *errno* set appropriately.

ERRORS

EINVAL The descriptor *hdl*, is invalid.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[Bn" (0x1b 0x5b 0x42 0x60).
It expects a response of the form "<ESC>[*P1*;*P2* .. *P6*R" (0x1b 0x5b *P1* 0x3b *P2* .. *P6* 0x52) where *P1* thru *P6* are the attribute states.

RESTRICTIONS

None

SEE ALSO

fpui_write_string(3fpui), **fpui_read_string**(3fpui), **fpui_get_auto_wrap**(3fpui), **fpui_get_auto_scroll**(3fpui), **fpui_get_auto_repeat**(3fpui), **fpui_get_backlight**(3fpui), **fpui_get_backlight_timeout**(3fpui)

NAME

fpui_get_window_size – Get the current window size

SYNOPSIS

#include <fpui.h>

int fpui_get_window_size(fpui_handle *hdl*, **int *** *row*, **int *** *column* **)**

DESCRIPTION

The **fpui_get_window_size**(3fpui) library call will return the current size of the actual Front panel Display module to the application.

Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
row	a reference of an integer where the number of rows will be stored.
column	a reference of an integer where the number of columns will be stored.

RETURN VALUE

fpui_get_window_size(3fpui) will return 0 (zero) upon success and -1 upon error with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

Should the actual Front Panel Display be removed, **fpui_get_window_size**(3fpui) will return (0, 0) as the current window size. The Front Panel hardware does not support this operation directly. The inquiry is serviced by the window which maintains this information.

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_home – Home cursor

SYNOPSIS

#include <fpui.h>

int fpui_home(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_home**(3fpui) library call will move the cursor to the upper left corner of the display, the home position, without affecting any other part of the display.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui)

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[H" (0x1b 0x5b 0x48).

RESTRICTIONS

None

SEE ALSO

fpui_write_string(3fpui)

NAME

fpui_open – Open a terminal interface

SYNOPSIS

#include <fpui.h>

fpui_handle fpui_open(int *flags*, **const char *** *regname* **)**

DESCRIPTION

The **fpui_open(**3fpui**)** library call is used to acquire a unique handle to a window which is an argument to most of the other commands in this fpui library. The parameter *flags* is one of **O_RDONLY**, **O_WRONLY,** or **O_RDWR** which request opening the window read-only, write-only, or read/write, respectively. Flags may also be or'd with O_DIRECT to bypass the window management and O_NONBLOCK to support read and write operations that will not block. When O_DIRECT is specified data is only sent when the window is in focus. In addition to acquiring the handle, the identifier *regname* is used to register this window with the Front Panel Manager. The name given in *regname* will appear in the Front Panel Manager Window selection screen.

RETURN VALUE

fpui_open(3fpui**)** returns the new descriptor, or -1 if an error occurred with *errno* set appropriately.

ERRORS

ENXIO	Underlying operation refers to an invalid device.
ENOMEM	Insufficient memory was available.
EACCES	There are too many windows already reseved to grant this request.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

The API provides a text-based user interface capability to allow application programs running concurrently on an ATC controller unit to share the controller's Front Panel display. The API provides up to 16 virtual display screens (referred to as "windows") that can be used by application programs as their user interface display. The display size of the windows is equal to physical display size (lines x characters) of the controller's Front Panel display (if one exists). The display size of the windows have a minimum size of 4 lines x 40 characters and a maximum size of 24 lines x 80 characters. If no physical display exists, the API operates as if it has a display with a size of 8 lines x 40 characters. Only one window is displayed at a time on the Front Panel display. When a window is displayed, the API displays the character representation of the window on the Front Panel display (if one exists). The application program associated with the window displayed receives the characters input from the Front Panel input device (Ex. keyboard or keypad). When an application window is displayed, it is said to have "focus". The window which has focus when the controller is powered up is called the "default window." The API supports the display character set as defined in the ATC Controller Standard, Section 7.1.4. Screen attributes described by the ATC Controller Standard, Section 7.1.4, are maintained for each window independently. Each window has separate input and output buffers unique from other windows.

The API user interface capability two modes of operation controlled by **O_DIRECT**. When **O_DIRECT** is not used, the API buffers I/O when a window is out of focus, maps character sequences to key codes, etc. using a "virtual terminal." This allows application programs to act on their window as if it were in focus even when it is not. When **O_DIRECT** is used, it is the responsibility of the calling program to maintain the window's appearance when it is refreshed or

goes in/out focus. **O_DIRECT** should be used when an application wants to use raw (uninterpreted) reads/writes or use graphics commands (as described in the ATC Controller Standard, Section 7.1.4). For these cases, **O_DIRECT** is used to indicate to the API to bypass the virtual terminal facility. If a window is opened using **O_DIRECT** and it receives write commands or graphics commands while not in focus, then the API ignores the commands.

The API provides a window selection screen called the Front Panel Manager Window from which Operational Users may select a window to have focus. Application names associated with each window are listed. The application names are limited to 16 characters. If there is no application program associated with a window, the window number is listed with a blank application name. The default Front Panel Manager Window size is 8 lines x 40 characters.

The API provides a window called the ATC Configuration Window. Operational Users are able to view ATC configuration information on this window that is provided by the Linux uname(2) function (ATC Controller Standard, Section 2.2.5), the API's version number(s), and the ATC Host Module EEPROM information (ATC Controller Standard, Annex B). The default ATC Configuration Window size is 8 lines x 40 characters.

RESTRICTIONS

None

SEE ALSO

fpui_close(3fpui)

NAME

fpui_open_aux_switch – Open the Aux Switch interface

SYNOPSIS

```
#include        <fpui.h>

int fpui_open_aux_switch( )
```

DESCRIPTION

The **fpui_open_aux_switch(**3fpui**)** library call is used to reserve exclusive access to the Aux Switch. One, and only one, process may hold the reservation at a time.

RETURN VALUE

fpui_open_aux_switch(3fpui) returns 0 (zero) upon success or -1 if an error occurred with *errno* set appropriately.

ERRORS

ENOMEM	Insufficient memory was available.
EACCES	The request access to the underlying device or object is not allowed. This could occur if the Aux Switch is currently opened by another process.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_read_aux_switch(3fpui**), fpui_close_aux_switch(**3fpui**)**

NAME

fpui_poll – Poll for presence of data

SYNOPSIS

#include <fpui.h>

boolean fpui_poll(fpui_handle *hdl*, int *flags* **)**

DESCRIPTION

The **fpui_poll**(3fpui) library call will check for the presence of data from the device associated with the specified descriptor.
Parameters:
hdl is the descriptor returned by **fpui_open**(3fpui).
flags is O_NONBLOCK if the call should return immediately.

RETURN VALUE

fpui_poll(3fpui) will return TRUE if data is available, FALSE if non-blocking and no data is available, or -1 if an error occurred with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EINTR	A signal occurred before any requested event.
ENOMEM	There is not enough memory available for this operation.

NOTES

None

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_read – Read from the Front Panel device

SYNOPSIS

#include <fpui.h>

ssize_t fpui_read(fpui_handle *hdl*, **char *** *buf*, **int** *size* **)**

DESCRIPTION

The **fpui_read**(3fpui) library call attempts to read at most *size* bytes of information from *hdl* into the buffer referenced by *buf*.

RETURN VALUE

fpui_read(3fpui) will return the number of bytes actually transferred to the buffer *buf*, or -1 if an error occurred with *errno* set appropriately.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for reading; or the object was opened with the **O_DIRECT** flag, and either the address specified in *buf* or the value specified in *count* is not suitably aligned.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_read_char(3fpui), **fpui_read_string**(3fpui)

NAME

fpui_read_aux_switch – Read from the Aux Switch device

SYNOPSIS

```
#include        <fpui.h>

boolean fpui_read_aux_switch( )
```

DESCRIPTION

The **fpui_read_aux_switch(**3fpui**)** library call will return TRUE if the Aux Switch is on and FALSE if the SWITCH is off.

RETURN VALUE

fpui_read_aux_switch(3fpui**)** will return TRUE if the Aux Switch is on, FALSE if the Aux Switch is off, or -1 if an error occurred with *errno* set appropriately.

ERRORS

EBADF	The underlying object or device used to access the Aux Switch is invalid.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	The underlying object or device used to access the Aux Switch is unsuitable for this operation.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_open_aux_switch(3fpui**), fpui_close_aux_switch(**3fpui**)**

NAME

fpui_read_char – Read one character from the Front Panel device

SYNOPSIS

#include <fpui.h>

char fpui_read_char(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_read_char**(3fpui) library call attempts to read at most one character of information from the object referenced by *hdl*.

RETURN VALUE

fpui_read_char(3fpui) will return either the character read, or -1 if an error occurred, with *errno* set appropriately.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.
EBADF	*hdl* is not a valid descriptor.
EFAULT	The underlying buffer is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for reading; or the object was opened with the **O_DIRECT** flag, and either the address specified in *buf* or the value specified in *count* is not suitably aligned.

NOTES

This routine may be implemented by wrapping a **fpui_read**(3fpui) operation.

RESTRICTIONS

None

SEE ALSO

fpui_read(3fpui), **fpui_read_string**(3fpui)

NAME

fpui_read_string – Read from the Front Panel device and NULL terminate the string

SYNOPSIS

#include <fpui.h>

ssize_t fpui_read_string(fpui_handle *hdl*, **char *** *buf*, **int** *size* **)**

DESCRIPTION

The **fpui_read_string**(3fpui) library call attempts to read at most *size* minus 1 bytes of information from the object referenced by *hdl* into the buffer referenced by *buf*. The string is then **NULL** terminated before returning.

RETURN VALUE

fpui_read_string(3fpui) will return the number of bytes actually transferred to the buffer *buf* (not including the termination character) or -1 if an error occurred, with *errno* set appropriately.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and no data was immediately available for reading.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for reading; or the object was opened with the **O_DIRECT** flag, and either the address specified in *buf* or the value specified in *count* is not suitably aligned.

NOTES

This routine may be implemented by wrapping a **fpui_read**(3fpui) operation.

RESTRICTIONS

None

SEE ALSO

fpui_read(3fpui), **fpui_read_char**(3fpui)

NAME

fpui_refresh – Refresh the Front Panel Display

SYNOPSIS

#include <fpui.h>

int fpui_refresh(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_refresh**(3fpui) library call will clear the Front Panel Display and redraw it according to the contents of the window. This operation has no effect if the calling applications window is not in focus.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

The Front Panel hardware does not support this operation directly. The inquiry is serviced by the window which then carries out the request.

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_reset_all_attributes – Reset all attributes to their off state

SYNOPSIS

#include <fpui.h>

int fpui_reset_all_attributes(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_reset_all_attributes**(3fpui) library call will turn off or disable the following attributes:

Character Blink
Cursor Blink
Reverse Video
Underline

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui)

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[0m" (0x1b 0x5b 0x30 0x6d)

RESTRICTIONS

None

SEE ALSO

fpui_write_string(3fpui)

NAME

fpui_reset_keymap – Reset and clear the entire keymap list

SYNOPSIS

#include <fpui.h>

int fpui_reset_keymap(fpui_handle *hdl*, **int** *type* **)**

DESCRIPTION

The **fpui_reset_keymap**(3fpui) library call is used to clear the entire keymap or to reset it to a default mapping.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui)

type when 0 removes all mappings
when 1 removes all mappings and preloads default mappings.

RETURN VALUE

On success, zero (0) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui)

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_set_keymap(3fpui), **fpui_get_keymap**(3fpui), **fpui_del_keymap**(3fpui)

NAME

fpui_set_auto_repeat – Set the auto repeat state

SYNOPSIS

#include <fpui.h>

int fpui_set_auto_repeat(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_auto_repeat**(3fpui) library call will set auto repeat mode according to the value of *state*. When auto repeat is enabled, all key strokes from the front panel will repeat as long as the key is held depressed. When this mode is not enables there will be only one key code sent per key press.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set auto repeat mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[08h" (0x1b 0x5b 0x30 0x38 0x68) to assert the mode and "<ESC>[08l" (0x1b 0x5b 0x30 0x38 0x6c) to negate the mode.

RESTRICTIONS

None

SEE ALSO

fpui_get_attributes(3fpui), **fpui_set_auto_scroll**(3fpui), **fpui_set_auto_wrap**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_auto_scroll – Get the auto scroll state

SYNOPSIS

#include <fpui.h>

int fpui_set_auto_scroll(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_auto_scroll**(3fpui) library call will set auto repeat mode according to the value of *state*. Auto scroll mode controls the display response when the cursor is on the bottom most line of the display and a new line is sent. If auto scroll is **TRUE**, the display will scroll all lines on the display up one line, overwriting the top most line. If auto scroll is **FALSE,** the display remains intact but the cursor will move to the top most line. In either case the column position of the cursor is not effected by the new line.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set auto repeat mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[<47h" (0x1b 0x5b 0x3c 0x34 0x37 0x68) to assert the mode and “<ESC>[<47l” (0x1b 0x5b 0x3c 0x34 0x37 0x6c) to negate the mode.

RESTRICTIONS

None

SEE ALSO

fpui_get_attributes(3fpui), **fpui_set_auto_repeat**(3fpui), **fpui_set_auto_wrap**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_auto_wrap – Set the auto wrap state

SYNOPSIS

#include <fpui.h>

int fpui_set_auto_wrap(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_auto_wrap**(3fpui) library call will set auto wrap mode according to the value of *state*. Auto wrap mode controls the displays response when the cursor is in the right most column and a character is sent. If auto wrap mode is **TRUE,** the character is displayed at the current cursor position and the cursor is wrapped back to the beginning of the line. If auto wrap mode is **FALSE,** the character is displayed at the current cursor position and the cursor will remain in the right most column. Any subsequent characters will overwrite the previous ones.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set auto wrap mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[07h" (0x1b 0x5b 0x30 0x37 0x68) to assert the mode and "<ESC>[07l" (0x1b 0x5b 0x30 0x37 0x6c) to negate the mode.

RESTRICTIONS

None

SEE ALSO

fpui_get_attributes(3fpui), **fpui_set_auto_repeat**(3fpui), **fpui_set_auto_scroll**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_backlight – Set the current state of the Backlight

SYNOPSIS

#include <fpui.h>

int fpui_set_backlight(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_backlight**(3fpui) library call will turn the backlight on or off as specified by the argument *state*. When turning the backlight on, it will remain on as long as the time between key strokes does not exceed the timeout value. When turning the backlight off, it will go off immediately, if the window associated with the request is in focus.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set the backlight to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned and *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[<5h" (0x1b 0x5b 0x3c 0x35 0x68) to illuminate the backlight and "<ESC>[<5l" (0x1b 0x5b 0x3c 0x35 0x6c) to extinguish the backlight.

RESTRICTIONS

None

SEE ALSO

fpui_get_backlight(3fpui), **fpui_get_attributes**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_backlight_timeout – Set the timeout value of the Backlight

SYNOPSIS

#include <fpui.h>

int fpui_set_backlight_timeout(fpui_handle *hdl*, **int** *timeout* **)**

DESCRIPTION

The **fpui_set_backlight_timeout**(3fpui) library call will set the backlight timeout value for the window associated with the calling application. This value is used only when the window remains in focus for a period longer than the timeout value. If this applications window is not in focus (or looses focus), this value is stored but has no other effect.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

timeout is the number of seconds to keep the backlight illuminated.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[<*Pn*S" (0x1b 0x5b 0x3c *Pn* 0x53). Where *Pn* is the number of seconds to keep the backlight illuminated.

RESTRICTIONS

None

SEE ALSO

fpui_get_backlight(3fpui), **fpui_set_backlight**(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_character_blink – Set the current character blink mode state

SYNOPSIS

#include <fpui.h>

int fpui_set_character_blink(fpui_handle *hdl,* **boolean** *state* **)**

DESCRIPTION

The **fpui_set_character_blink**(3fpui) library call will assert or negate character blink mode. The current character blink attribute is applied to all characters written to the display after the attribute has been set. To blink a single word one would assert character blink mode, write the entire work and finally negate character blink mode.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set the character blink mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[25h" (0x1b 0x5b 0x32 0x35 0x68) to assert character blink mode and "<ESC>[25l" (0x1b 0x5b 0x32 0x35 0x6c) to negate character blink mode.

RESTRICTIONS

None

SEE ALSO

fpui_get_character_blink(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_cursor – Set the cursor state

SYNOPSIS

#include <fpui.h>

int fpui_set_cursor(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_cursor**(3fpui) library call will set the state of the cursor as specified by the *state* parameter. When *state* is **TRUE,** the cursor will be visible. When *state* is **FALSE,** the cursor will not be visible. Even though the cursor is not visible, it still exists logically.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set the cursor mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[025h" (0x1b 0x5b 0x30 0x32 0x35 0x68) to make the cursor visible and "<ESC>[025l" (0x1b 0x5b 0x30 0x32 0x35 0x6c) to make the cursor invisible.

RESTRICTIONS

None

SEE ALSO

fpui_get_cursor(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_cursor_blink – Set the cursor blink mode state

SYNOPSIS

#include <fpui.h>

int fpui_set_cursor_blink(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_cursor_blink**(3fpui) library call will set the blink mode for the cursor. When state is **TRUE** the cursor will be set to blink, when **FALSE** the cursor will be on solid. This mode does not override or affect the mode set by fpui_set_cursor(3fpui).

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set the cursor mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned and *errno* is set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[33h" (0x1b 0x5b 0x33 0x33 0x68) to make the cursor visible and "<ESC>[33l" (0x1b 0x5b 0x33 0x33 0x6c) to make the cursor invisible.

RESTRICTIONS

None

SEE ALSO

fpui_get_cursor_blink(3fpui)

NAME

fpui_set_cursor_pos – Set the current Cursor Position

SYNOPSIS

#include <fpui.h>

int fpui_set_cursor_pos(fpui_handle *hdl*, **int** *row*, **int** *column* **)**

DESCRIPTION

The **fpui_set_cursor_pos**(3fpui) library call will set the cursors position to the location specified by *row* and *column*. If either *row* or *column* extend beyond the bounds of the physical display, the cursor will be set to the last valid location in the specified direction.

Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
row	is the row index the cursor should move to.
column	is the column index the cursor should move to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned and *errno* is set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[*Py*;*Px*f" (0x1b 0x5b *Py* 0x3b *Px* 0x66) where *Py* and *Px* are the *row* and *column* parameters.

RESTRICTIONS

Row and *column* values are truncated to the limits of the physical display.

SEE ALSO

fpui_get_cursor_pos(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_emergency – Set or Reset emergency mode for this application

SYNOPSIS

#include <fpui.h>

int fpui_set_emergency(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_emergency**(3fpui) library call will assert or negate emergency mode for this applications window (when *state* is **TRUE**). Emergency mode is indicated by having the front panel backlight flash and the registered name of the application be displayed with both reverse video and character blink asserted. This mode is automatically cleared when the application achieves focus. The application can also clear the mode by passing a **FALSE** state to the call.
Parameters:

hdl	is the descriptor returned by fpui_open(3fpui).
state	the value to set emergency mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned and *errno* is set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request is not valid.

NOTES

While each application can assert and negate its own emergency mode, the display will continue to flash as long as any application has emergency mode asserted. The registered name, however, will track the individual applications assertion or negation of emergency mode.

RESTRICTIONS

None

SEE ALSO

None

NAME

fpui_set_keymap – Define an escape sequence mapping

SYNOPSIS

#include <fpui.h>

int fpui_set_keymap(fpui_handle *hdl*, **char** *key*, **char *** *eseq* **)**

DESCRIPTION

The **fpui_set_keymap**(3fpui) library call is used to add an escape sequence mapping to the calling process' window. Operationally, when an escape sequence is received by the window, typically from the keypad, it is compared to the list of key mappings. If a match is found, the entire escape sequence is replaced with the single *key* character and returned to the caller. Mappings only affect character strings moving from the front panel hardware to the application in focus.
Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

key is the single byte code to be returned when the character sequence is found in the received character string.

eseg is the null terminated sting containing the incoming sequence to be replaced. Only the first 7 bytes are used.

RETURN VALUE

On success, zero (0) is returned. On error, -1 is returned and *errno* is set appropriately.

ERRORS

ENOMEM More then 16 mappings are defined.

NOTES

There are no checks for duplicate *key* values or duplicate escape sequences. Escape sequences are compared in the order they appear in the list. New *key* mappings are inserted in the first available slot in the list, which may be a slot vacated by an earlier delete operation. This means that the list may not be in the order mappings were added. It should be also noted that there is nothing prohibiting an application from overriding the standard ATC escape sequences.

RESTRICTIONS

None

SEE ALSO

fpui_get_keymap(3fpui), **fpui_del_keymap**(3fpui), **fpui_reset_keymap**(3fpui)

NAME

fpui_set_led – Set the state of the status LED

SYNOPSIS

#include <fpui.h>

int fpui_set_led(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_led**(3fpui) library call will set the state of the status LED to the value specified by *state*. This state is maintained by the window and will be reflected in the actual Front Panel LED when the application window gains focus.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned and *errno* is set appropriately.

ERRORS

EBADF	*hdl* is not a valid descriptor.
EFAULT	A reference to an inaccessible memory area was attempted.
EINVAL	Request not valid.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_get_led(3fpui)

NAME

fpui_set_reverse_video – Set the state of reverse video mode

SYNOPSIS

#include <fpui.h>

int fpui_set_reverse_video(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_set_reverse_video**(3fpui) library call will assert or negate reverse video mode. The current reverse video attribute is applied to all characters written to the display after the attribute has been set. To reverse video a single, assert reverse video mode, write the entire work and finally negate reverse video mode.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set the reverse video mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned and *errno* is set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[27h" (0x1b 0x5b 0x32 0x37 0x68) to assert reverse video mode and "<ESC>[27l" (0x1b 0x5b 0x32 0x37 0x6c) to negate reverse video mode.

RESTRICTIONS

None

SEE ALSO

fpui_get_reverse_video(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_tab – Set a Stop Tab at the current cursor position

SYNOPSIS

#include <fpui.h>

int fpui_set_tab(fpui_handle *hdl* **)**

DESCRIPTION

The **fpui_set_tab**(3fpui) library call will define a stop tab in the column the cursor currently occupies.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>H" (0x1b 0x48).

RESTRICTIONS

None

SEE ALSO

fpui_clear_tab(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_underline – Set the state of underline mode

SYNOPSIS

#include <fpui.h>

int fpui_set_underline(fpui_handle *hdl*, **boolean** *state* **)**

DESCRIPTION

The **fpui_get_underline**(3fpui) library call will assert or negate underline mode. The current underline attribute is applied to all characters written to the display after the attribute has been set. To underline a single word, assert underline mode, write the entire work and finally negate underline mode.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

state the value to set the underline mode to.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This operation may be implemented by calling **fpui_write_string**(3fpui) with the constant string "<ESC>[24h" (0x1b 0x5b 0x32 0x34 0x68) to assert underline mode and "<ESC>[24l" (0x1b 0x5b 0x32 0x34 0x6c) to negate underline mode.

RESTRICTIONS

None

SEE ALSO

fpui_get_underline(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_set_window_attr – Set the current Window attributes

SYNOPSIS

#include <fpui.h>

int fpui_set_window_attr(fpui_handle *hdl*, **unsigned int** *arg* **)**

DESCRIPTION

The **fpui_set_window_attr**(3fpui) library call will set the window attributes according to the values passed by parameter *arg*. The bitfields of *arg* are encoded as follows:

```
union {
        int             errcode;
        unsigned int    auto_wrap:1,
                        auto_scroll:1,
                        auto_repeat:1,
                        backlight:1,
                        :4,
                        backlight_timeout:8,
                        :16;
}
```

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

arg the integer representation of the union described above.

RETURN VALUE

On success, 0 (zero) is returned. On error, -1 is returned with *errno* set appropriately.

ERRORS

Refer to **fpui_write_string**(3fpui) for additional errors.

NOTES

This call decodes the encoded bit fields and makes individual calls to set attributes as needed. The *errcode* field is unused in this operation.

RESTRICTIONS

None

SEE ALSO

fpui_write_string(3fpui), **fpui_set_auto_wrap**(3fpui), **fpui_set_auto_scroll**(3fpui), **fpui_set_auto_repeat**(3fpui), **fpui_set_backlight**(3fpui), **fpui_set_backlight_timeout**(3fpui)

NAME

fpui_write – Write to the Front Panel device

SYNOPSIS

#include <fpui.h>

ssize_t fpui_write(fpui_handle *hdl*, **char *** *buf*, **int** *size* **)**

DESCRIPTION

The **fpui_write**(3fpui) library call attempts to write *size* bytes of information from the buffer referenced by *buf* to the object referenced by *hdl*.
Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).
buf the reference to the data to be written.
size the number of bytes to write.

RETURN VALUE

fpui_write(3fpui) will return the number of bytes actually transferred to the device at *hdl*, or -1 if an error occurred, with *errno* set appropriately.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for writing; or the object was opened with the **O_DIRECT** flag and either the address specified in buf or the value specified in count is not suitably aligned.
EIO	A low-level I/O error occurred.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_write_char(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_write_at – Write to the Front Panel device at a specified location

SYNOPSIS

#include <fpui.h>

ssize_t fpui_write_at(fpui_handle *hdl*, **char *** *buf*, **int** *size*, **int** *row*, **int** *column* **)**

DESCRIPTION

The **fpui_write**(3fpui) library call first moves the cursor to the location specified by *row* and *column* and then attempts to write *size* bytes of information from the buffer referenced by *buf* to the device referenced by *hdl*.

Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
buf	the reference to the data to be written.
size	the number of bytes to write.
row	the cursor row index.
column	the cursor column index.

RETURN VALUE

fpui_write_at(3fpui) will return the number of bytes actually transferred to the device at *hdl* or -1 if an error occurred with *errno* set appropriately.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for writing; or the object was opened with the **O_DIRECT** flag and either the address specified in buf or the value specified in count is not suitably aligned.
EIO	A low-level I/O error occurred.

NOTES

None

RESTRICTIONS

Row and *column* values are truncated to the limits of the physical display.

SEE ALSO

fpui_write(3fpui), **fpui_set_cursor_pos**(3fpui)

NAME

fpui_write_char – Write a single character to the Front Panel device

SYNOPSIS

#include <fpui.h>

ssize_t fpui_write_char(fpui_handle *hdl*, **char** *ch* **)**

DESCRIPTION

The **fpui_write_char**(3fpui) library call attempts to write a single character *ch* to the object referenced by *hdl*.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

ch the character to be written.

RETURN VALUE

fpui_write_char(3fpui) will return the number of bytes actually transferred to the device at *hdl* or -1 if an error occurred with *errno* set appropriately.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for writing; or the object was opened with the **O_DIRECT** flag and either the address specified in buf or the value specified in count is not suitably aligned.
EIO	A low-level I/O error occurred.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_write(3fpui), **fpui_write_string**(3fpui)

NAME

fpui_write_char_at – Write a single character to the Front Panel device at a specified location

SYNOPSIS

#include <fpui.h>

ssize_t fpui_write_char_at(fpui_handle *hdl*, **char** *ch*, **int** *row*, **int** *column* **)**

DESCRIPTION

The **fpui_write_char_at**(3fpui) library call first moves the cursor to the location specified by *row* and *column* and then attempts to write a single character *ch* to the device referenced by *hdl*.
Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
ch	the character to be written.
row	the cursor row index.
column	the cursor column index.

RETURN VALUE

fpui_write_char_at(3fpui) will return the number of bytes actually transferred to the device at *hdl* (in this case, 1) or -1 if an error occurred with *errno* set appropriately.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for writing; or the object was opened with the **O_DIRECT** flag and either the address specified in buf or the value specified in count is not suitably aligned.
EIO	A low-level I/O error occurred.

NOTES

None

RESTRICTIONS

Row and *column* values are truncated to the limits of the physical display.

SEE ALSO

fpui_write_char(3fpui), **fpui_set_cursor_pos**(3fpui)

NAME

fpui_write_string – Write a NULL terminated string to the Front Panel device

SYNOPSIS

#include <fpui.h>

ssize_t fpui_write_string(fpui_handle *hdl*, **char *** *str* **)**

DESCRIPTION

The **fpui_write_string**(3fpui) library call attempts to write the string referenced by *str* to the object referenced by *hdl*.

Parameters:

hdl is the descriptor returned by **fpui_open**(3fpui).

str the reference to the NULL terminated string to be written.

RETURN VALUE

fpui_write_string(3fpui) will return the number of bytes actually transferred to the device at *hdl*,or -1 if an error occurred with *errno* set appropriately. The terminating NULL character is not written to the device.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for writing; or the object was opened with the **O_DIRECT** flag and either the address specified in buf or the value specified in count is not suitably aligned.
EIO	A low-level I/O error occurred.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fpui_write(3fpui), **fpui_write_char**(3fpui)

NAME

fpui_write_string_at – Write a NULL terminated string to the Front Panel device at a specified location

SYNOPSIS

#include <fpui.h>

ssize_t fpui_write_string_at(fpui_handle *hdl*, **char *** *str*, **int** *row*, **int** *column* **)**

DESCRIPTION

The **fpui_write_string_at**(3fpui) library call first moves the cursor to the location specified by *row* and *column* and then attempts to write the string referenced by *str* to the object referenced by *hdl*. Parameters:

hdl	is the descriptor returned by **fpui_open**(3fpui).
str	the reference to the NULL terminated string to be written.
row	the cursor row index.
column	the cursor column index.

RETURN VALUE

fpui_write_string_at(3fpui) will return the number of bytes actually transferred to the device at *hdl*, or -1 if an error occurred, with *errno* set appropriately. The terminating NULL character is not written to the device.

ERRORS

EAGAIN	Non-blocking I/O has been selected using **O_NONBLOCK** and the write would block.
EBADF	*hdl* is not a valid descriptor.
EFAULT	*buf* is outside your accessible address space.
EINTR	The call was interrupted by a signal before any data was read.
EINVAL	*hdl* is attached to an object which is unsuitable for writing; or the object was opened with the **O_DIRECT** flag and either the address specified in buf or the value specified in count is not suitably aligned.
EIO	A low-level I/O error occurred.

NOTES

None

RESTRICTIONS

Row and *column* values are truncated to the limits of the physical display.

SEE ALSO

fpui_write_string(3fpui), **fpui_set_cursor_pos**(3fpui)

4.2 Field I/O Manager Functions

This section specifies the API Field I/O Manager functions. They are listed alphabetically with each specification beginning on a new page. Operationally, they can be grouped as shown below.

- General Functions – fio_register, fio_deregister, fio_fiod_register, fio_fiod_deregister, fio_fiod_enable, fio_fiod_disable, fio_query_fiod, fio_fiod_status_get, fio_fiod_status_reset and fio_apiver.
- Input Configuration Functions – fio_fiod_inputs_get, fio_fiod_inputs_filter_set and fio_fiod_inputs_filter_get.
- Output Configuration Functions – fio_fiod_outputs_set, fio_fiod_outputs_get, fio_fiod_outputs_reservation_set and fio_fiod_outputs_reservation_get.
- Frame Functions – fio_fiod_frame_schedule_set, fio_fiod_frame_schedule_get, fio_fiod_frame_size, fio_fiod_frame_read, fio_fiod_frame_notify_register, fio_fiod_frame_notify_deregister and fio_query_frame_notify_status.
- Transition Buffer Functions – fio_fiod_inputs_trans_set, fio_fiod_inputs_trans_get and fio_fiod_inputs_trans_read.
- Watchdog/Health Monitor Functions – fio_fiod_wd_register, fio_fiod_wd_deregister, fio_fiod_wd_reservation_set, fio_fiod_wd_reservation_get, fio_fiod_wd_heartbeat, fio_hm_register, fio_hm_deregister, fio_hm_heartbeat and fio_hm_fault_reset.
- Fault/Volt Monitor Functions – fio_fiod_ts_fault_monitor_set, fio_fiod_ts_fault_monitor_get, fio_fiod_ts1_volt_monitor_set and fio_fiod_ts1_volt_monitor_get.
- CMU/MMU/Channel Functions – fio_fiod_cmu_fault_set, fio_fiod_cmu_fault_get, fio_fiod_cmu_dark_channel_set, fio_fiod_cmu_dark_channel_get, fio_fiod_mmu_flash_bit_set, fio_fiod_mmu_flash_bit_get, fio_fiod_channel_reservation_set, fio_fiod_channel_reservation_get, fio_fiod_channel_map_set, fio_fiod_channel_map_count and fio_fiod_channel_map_get.

The header file, “fio.h,” defines all the function calls in a library named “libfio.so.”

NAME

fio_apiver – Obtain version information about the FIO API

SYNOPSIS

```
#include        <fio.h>

char *fio_apiver( FIO_APP_HANDLE        app_handle,
                  FIO_VERSION           which )
```

DESCRIPTION

fio_apiver(3fio) is used to obtain version information about the FIO API.

app_handle is a **FIO_APP_HANDLE** returned from a previously successful call to **fio_register**(3fio). *which* is an indication as to the version desired. Valid values for *which* are: **FIO_VERSION_LIBRARY** and **FIO_VERSION_LKM**, to obtain a version stamp for the FIO API Library and the FIO API Loadable Kernel Module (LKM) respectively.

A character string containing the version stamp is returned.

RETURN VALUES

Upon successful completion, a character string containing the version stamp is returned. A NULL return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The character string buffer returned is owned by the FIO API and must NOT be freed by the application program.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio)

NAME

fio_deregister – Deregister with the FIO API

SYNOPSIS

#include <fio.h>

int fio_deregister(FIO_APP_HANDLE *app_handle* **)**

DESCRIPTION

fio_deregister(3fio) is used to deregister this application program from using all FIO API services.

app_handle is a **FIO_APP_HANDLE** returned from a previously successful call to **fio_register**(3fio).

This function cleans up, releases any memory associated with this application program and closes all services that may have been registered and in use by the application program. All service handles are invalidated. There is no need for the application program to deregister individual services; this call will deregister ALL currently active FIO API services for this application program.

app_handle is invalidated and may no longer be used. All **FIO_DEV_HANDLE**s and FIO API service handles are disabled, deregistered and invalidated, and may no longer be used. All system values set by this application program are set to the next precedence level, set by other application programs; as determined by individual service algorithms (see associated MAN pages for specifics).

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL One or more arguments are invalid.

NOTES

When an application program exits or terminates for any reason, the FIO API automatically calls **fio_deregister**(3fio) to clean up the FIO API for this application.

All system values set by this application program are set to the next precedence level, set by other application programs, as determined by individual service algorithms (see associated MAN pages for specifics).

When an application program deregisters for access to FIO services, the FIO API deregisters all FIODs registered by that application program, as if **fio_fiod_deregister**(3fio) had been called for each one.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_deregister**(3fio), **fio_hm_deregister**(3fio), **fio_fiod_wd_deregister**(3fio)

NAME

fio_fiod_channel_map_count – Return the count of the current active channel maps

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_channel_map_count( FIO_APP_HANDLE app_handle,
                                FIO_DEV_HANDLE  dev_handle,
                                FIO_VIEW        view )
```

DESCRIPTION

This function is used to return the count of the current active channel mappings for the specified FIOD. This count can then be utilized by an application program to allocate a buffer large enough for a successful **fio_fiod_channel_map_get**(3fio) call.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call, for a **FIOMMU** or **FIOCMU** device. *view* is an indication as to the view of channel map count to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively.

RETURN VALUES

Upon successful completion, a count of the number of active channel maps is returned for the *view* indicated. This value may be utilized to allocate buffers for a successful **fio_fiod_channel_map_get**(3fio) call. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

It is possible for the **FIO_VIEW_SYSTEM** *view* of the returned count to change in the system prior to calling **fio_fiod_channel_map_get**(3fio); as another application program may run and change the state of the system between calling **fio_fiod_channel_map_count**(3fio) and **fio_fiod_channel_map_get**(3fio). The application program must handle this situation accordingly.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_outputs_reservation_get**(3fio), **fio_fiod_channel_reservation_set**(3fio), **fio_fiod_channel_reservation_get**(3fio), **fio_fiod_channel_map_set**(3fio), **fio_fiod_channel_map_get**(3fio)

NAME

fio_fiod_channel_map_get – Get the current active channel mappings

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_channel_map_get( FIO_APP_HANDLE        app_handle,
                              FIO_DEV_HANDLE        dev_handle,
                              FIO_VIEW              view,
                              FIO_CHANNEL_MAP       *channel_map,
                              unsigned int          count )
```

DESCRIPTION

This method is used to retrieve the current active channel mappings for the indicated **FIOMMU** or **FIOCMU** device.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call, of a **FIOMMU** or **FIOCMU** device. *view* is an indication as to the view of channel map to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *channel_map* is a pointer to an array of **FIO_CHANNEL_MAP** structures that will be filled in upon successful completion. *count* is a count of the number of *channel_map* elements passed, as returned by a successful **fio_fiod_channel_map_count**(3fio) call.

FIO_CHANNEL_MAP is defined as:

```
struct
{
        /* FROM */
        unsigned int        output_point;   /* Output Point Number */
        FIO_DEV_HANDLE      fiod;           /* Device of Output Point Number */

        /* TO */
        unsigned int        channel;        /* Channel Number */
        FIO_COLOR           color;          /* Green, yellow or red */
} fio_channel_map;
typedef struct fio_channel_map FIO_CHANNEL_MAP;
```

output_point and *fiod* are the definitions of a previously reserved output point on the designated FIOD; this constitutes the mapping “from” address. *channel* is a previously reserved channel. *color* is a color designation on this *channel*. The *channel* / *color* combination is for the **FIOMMU** or **FIOCMU** referenced by *dev_handle*. This structure is filled in and returned by this function. *fiod* is directly usable by this application program when **FIO_VIEW_APP** is passed. For a view of **FIO_VIEW_SYSTEM**, *fiod* is not directly usable.

Up to *count* channel mappings are returned. The *channel_map* array is a “positive” list of the channel mappings currently active for this application program (**FIO_VIEW_APP**) or system (**FIO_VIEW_SYSTEM**).

RETURN VALUES

Upon successful completion the count of the number of *channel_map* structures filled is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

Upon successful return, the *channel_map* array will contain the actual channel map values currently being utilized by the FIO API; for the *view* indicated.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

Channel to output point mappings that may be made are specific per unique FIOD.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_outputs_reservation_get**(3fio), **fio_fiod_channel_reservation_set**(3fio), **fio_fiod_channel_reservation_get**(3fio), **fio_fiod_channel_map_count**(3fio), **fio_fiod_channel_map_set**(3fio)

NAME

fio_fiod_channel_map_set – Map a reserved output point to a reserved channel/color

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_channel_map_set( FIO_APP_HANDLE      app_handle,
                              FIO_DEV_HANDLE      dev_handle,
                              FIO_CHANNEL_MAP     *channel_map,
                              unsigned int        count )
```

DESCRIPTION

This function provides a method for application programs to map/unmap reserved output points to reserved channels and colors on a registered **FIOMMU** or **FIOCMU** device

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call, of a **FIOMMU** or **FIOCMU** device. *channel_map* is a pointer to an array of **FIO_CHANNEL_MAP** structures. *count* is a count of the number of *channel_map* elements passed.

FIO_CHANNEL_MAP is defined as:

```
struct
{
        /* FROM */
        unsigned int        output_point;   /* Output Point Number */
        FIO_DEV_HANDLE      fiod;           /* Device of Output Point Number */

        /* TO */
        unsigned int        channel;        /* Channel Number */
        FIO_COLOR           color;          /* Green, yellow or red */
} fio_channel_map;
typedef struct fio_channel_map FIO_CHANNEL_MAP;
```

output_point and *fiod* are the definitions of a previously reserved output point on the designated FIOD; this constitutes the mapping “from” address. *channel* is a previously reserved channel. *color* is a color designation on this *channel*. The *channel* / *color* combination is for the **FIOMMU** or **FIOCMU** referenced by *dev_handle*.

Channel mapping is atomic; all or nothing. If any value in the *channel_map* array is found to be invalid, the entire operation will fail.

The *channel_map* array is a “positive” list of the channel mappings desired by this application program. To unmap a channel map, the appropriate *channel_map* array entry must be removed by an application program and **fio_fiod_channel_map_set**(3fio) must be called once again.

Any channel and color not mapped to an output point is be set to off.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
EPERM	A referenced output point or channel was not previously reserved.
ENOMEM	There is not enough memory available for this operation.

NOTES

Relinquishing a reserved output point or channel clears the channel map for the applicable output points.

The FIO API updates the mapped channel and color on a **FIOMMU** or **FIOCMU** device based on the value of the associated output point. For a **FIOMMU,** the output point plus and minus values are directly mapped to **FIOMMU** plus / minus. For a **FIOCMU**, the channel is set to off if, and only if, both plus and minus are set to off.

Channel to output point mappings that may be made are specific per unique FIOD.

The FIO API uses this mapping to set the contents of **FIOMMU** Frame 0 and **FIOCMU** Frames 61 and 67.

RESTRICTIONS

Only output points and channels previously reserved, utilizing **fio_fiod_outputs_reservation_set**(3fio) and **fio_fiod_channel_reservation_set**(3fio) respectively, may be mapped.

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_outputs_reservation_get**(3fio), **fio_fiod_channel_reservation_set**(3fio), **fio_fiod_channel_reservation_get**(3fio), **fio_fiod_channel_map_count**(3fio), **fio_fiod_channel_map_get**(3fio)

NAME

fio_fiod_channel_reservation_get – Get the reservation state of a **FIOMMU** or **FIOCMU** channel

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_channel_reservation_get( FIO_APP_HANDLE        app_handle,
                                      FIO_DEV_HANDLE        dev_handle,
                                      FIO_VIEW              view,
                                      unsigned char         *channel )
```

DESCRIPTION

This function is used to query the current reservation state of a **FIOMMU** or **FIOCMU** channel. Either the current application program's view or the current system view of channel reservation information may be retrieved.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call, for a **FIOCMU** or **FIOMMU**. *view* is an indication as to the view of channel reservation data to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *channel* is a pointer to an unsigned character buffer that will contain a bit array of the current reservation state of the channels; upon successful completion. This buffer must be at least **FIO_CHANNEL_BYTES** in length; no checking is performed by the function to ensure this. A 1 bit indicates that the corresponding channel is reserved, a 0 bit indicates that the corresponding channel is relinquished; not reserved. This bit array is a "positive" mask. The construction of this bit array is directly useable by **fio_fiod_channel_reservation_set**(3fio).

The macro **FIO_BIT_TEST**(*addr*, *num*) may be used to test for a bit being set in the resulting bit array; where *addr* is the address of the bit array (*channel*) and *num* is the bit number of the bit to be tested. No boundary checking is performed by this macro; the user must perform all boundary checking.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

Channels that may be reserved are specific per unique FIOD.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_channel_reservation_set**(3fio), **fio_fiod_channel_map_set**(3fio), **fio_fiod_channel_map_get**(3fio)

NAME

fio_fiod_channel_reservation_set – Set the reservation state of a **FIOMMU** or **FIOCMU** channel

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_channel_reservation_set( FIO_APP_HANDLE        app_handle,
                                      FIO_DEV_HANDLE        dev_handle,
                                      unsigned char         *channel )
```

DESCRIPTION

This function provides a method for application programs to reserve/relinquish exclusive (single application program) “write access” to a channel on a **FIOMMU** or **FIOCMU** device.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call, for a **FIOMMU** or **FIOCMU** device. *channel* is a pointer to an unsigned character buffer that contains a bit array of the desired reservation states of the channels, for a **FIOMMU** or **FIOCMU** device, for this application program. Channels that are to be reserved are set to 1 in this bit array. Channels that are to be relinquished (not reserved) are set to 0 in this bit array. This buffer must be at least **FIO_CHANNEL_BYTES** in length; no checking is performed by the function to ensure this. This bit array is a “positive” mask; the states of bits in this array are this application programs current view of the state (reserved/relinquished) of channels from its perspective.

If an application program attempts to reserve a channel that has already been reserved by that application program, it is not considered an error. If an application program relinquishes a channel that is already in the relinquished state for that application program, it is not considered an error. The bit array is a “positive” mask.

If a channel in a group of not already reserved channels cannot be reserved, the reservation attempt fails for the entire new group of channels; **ENOTTTY** is returned in *errno*. The reservation state of previously reserved channels is maintained; both the reserved and relinquished state of a channel. The reserve / relinquish operation is atomic; all or nothing.

The macros **FIO_BIT_SET**(*addr*, *num*) and **FIO_BIT_CLEAR**(*addr*, *num*) may be used to set and clear a bit in a bit array; where *addr* is the address of the bit array (*channel*) and *num* is the bit number of the bit to be set / cleared. No boundary checking is performed by these macros; the user must perform all boundary checking.

The macro **FIO_BITS_CLEAR**(*addr*, *size*) may be used to initialize a bit array to the 0 state; where *addr* is the address of the bit array (*channel*) and *size* is the size of the bit array in bytes (**FIO_CHANNEL_BYTES**).

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOTTY	One or more of the requested channels is already reserved by another application program.

NOTES

The FIO API allows only one application program to reserve write access to any individual channel. The FIO API allows multiple application programs to reserve different channels on a single **FIOMMU** or **FIOCMU**. The FIO API provides error codes so that the application program can determine if the reservation action was successful or if there was a conflict with another application program.

Relinquishing a reserved output point or channel clears the channel map for the applicable output points.

The FIO API makes channel reservations on a “first come first served basis.” Setting the *channel* bit array to all 0’s prior to this call will result in all reservations being relinquished.

The **fio_fiod_channel_reservation_get**(3fio) function returns the “positive” mask for either this application program’s or the current system reservations.

Channels that may be reserved are specific per unique FIOD.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_outputs_reservation_get**(3fio), **fio_fiod_channel_reservation_get**(3fio), **fio_fiod_channel_map_set**(3fio), **fio_fiod_channel_map_get**(3fio)

NAME

fio_fiod_cmu_dark_channel_get – Get the current **FIOCMU** Dark Channel Map

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_cmu_dark_channel_get( FIO_APP_HANDLE        app_handle,
                                   FIO_DEV_HANDLE        dev_handle,
                                   FIO_VIEW              view,
                                   FIO_CMU_DC_MASK       *mask )
```

DESCRIPTION

This function provides a method to get the Dark Channel Map selection for a registered **FIOCMU** FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOCMU** FIOD. *view* is an indication as to the view of *mask* to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *mask* is a pointer to **FIO_CMU_DC_MASK** that will contain the **FIOCMU** Dark Channel Map mask upon successful completion. Valid values are **FIO_CMU_DC_MASK1**, **FIO_CMU_DC_MASK2**, **FIO_CMU_DC_MASK3** and **FIO_CMU_DC_MASK4**.

The **FIO_VIEW_SYSTEM** *view* is the current Dark Channel Map mask being utilized by the system for this **FIOCMU** FIOD. The default *mask* is **FIO_CMU_DC_MASK1**.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The last Dark Channel Map mask set by all application programs is used by the system.

RESTRICTIONS

dev_handle must be a **FIOCMU** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_cmu_fault_set**(3fio), **fio_fiod_cmu_fault_get**(3fio), **fio_fiod_cmu_dark_channel_set**(3fio)

NAME

fio_fiod_cmu_dark_channel_set – Select a **FIOCMU** Dark Channel Map

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_cmu_dark_channel_set( FIO_APP_HANDLE      app_handle,
                                   FIO_DEV_HANDLE      dev_handle,
                                   FIO_CMU_DC_MASK     mask )
```

DESCRIPTION

This function provides a method to set the Dark Channel Map selection for a registered **FIOCMU** FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOCMU** FIOD. *mask* is a valid **FIO_CMU_DC_MASK** that contains the **FIOCMU** Dark Channel Map mask value. Valid values are **FIO_CMU_DC_MASK1**, **FIO_CMU_DC_MASK2**, **FIO_CMU_DC_MASK3** and **FIO_CMU_DC_MASK4**.

The last application program to set the Dark Channel Map mask is used by the system. The default Dark Channel Map mask is **FIO_CMU_DC_MASK1**.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

If multiple application programs attempt to set the Dark Channel Map selection, the FIO API uses the most recent selection.

RESTRICTIONS

dev_handle must be a **FIOCMU** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_cmu_fault_set**(3fio), **fio_fiod_cmu_fault_get**(3fio), **fio_fiod_cmu_dark_channel_get**(3fio)

NAME

fio_fiod_cmu_fault_get – Get the current FSA of a FIOCMU FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_cmu_fault_get( FIO_APP_HANDLE      app_handle,
                            FIO_DEV_HANDLE      dev_handle,
                            FIO_VIEW            view,
                            FIO_CMU_FSA         *fsa )
```

DESCRIPTION

This function provides a method to get the current fault state of a **FIOCMU** FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOCMU** FIOD. *view* is an indication as to the view of FSA to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *fsa* is a pointer to a **FIO_CMU_FSA** that will contain the **FIOCMU** *fsa* upon successful completion. The supported values are **FIO_CMU_FSA_NONE** (0), **FIO_CMU_FSA_NON_LATCHING** (1) and **FIO_CMU_FSA_LATCHING** (2). The precedence order is LATCHING, NON-LATCHING down to NONE.

The FSA **FIO_VIEW_SYSTEM** *view* is the current highest precedence for all application programs that have registered with the **FIOCMU** FIOD. The default FSA is **FIO_CMU_FSA_NONE** (0). If an application program sets the FSA to something other than NONE, it must subsequently set the FSA back to NONE to clear the fault condition.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

All **FIOCMU** registered application programs may set the FSA level. The value used by the FIO API is the current highest value for all registered application programs.

The **FIOCMU** must be enabled using fio_fiod_enable(3fio) in order for the FSA to become effective.

RESTRICTIONS

dev_handle must be a **FIOCMU** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_cmu_fault_set**(3fio), **fio_fiod_cmu_dark_channel_set**(3fio), **fio_fiod_cmu_dark_channel_get**(3fio)

NAME

fio_fiod_cmu_fault_set – Set the FSA of a FIOCMU FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_cmu_fault_set( FIO_APP_HANDLE      app_handle,
                            FIO_DEV_HANDLE      dev_handle,
                            FIO_CMU_FSA         fsa )
```

DESCRIPTION

This function provides a method to set the fault state of a **FIOCMU** FIOD to none, latching, or non-latching.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOCMU** FIOD. *fsa* is a **FIO_CMU_FSA**. The supported values are **FIO_CMU_FSA_NONE** (0), **FIO_CMU_FSA_NON_LATCHING** (1) and **FIO_CMU_FSA_LATCHING** (2). The precedence order is LATCHING, NON-LATCHING down to NONE.

The FSA value sent to the **FIOCMU** is the current highest precedence for all application programs that have registered with the **FIOCMU** FIOD. The default FSA is **FIO_CMU_FSA_NONE** (0). If an application program sets the FSA to something other than **FIO_CMU_FSA_NONE**, it must subsequently set the FSA back to **FIO_CMU_FSA_NONE** to clear the fault condition. The system FSA value sent to the **FIOCMU** is **FIO_CMU_FSA_NONE** if, and only if, all **FIOCMU** registered application programs set the FSA to **FIO_CMU_FSA_NONE**.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

All **FIOCMU** registered application programs may set the FSA level. The value used by the FIO API is the current highest value for all registered application programs.

The **FIOCMU** must be enabled using **fio_fiod_enable**(3fio) in order for the FSA to become effective.

RESTRICTIONS

dev_handle must be a **FIOCMU** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_cmu_fault_get**(3fio) , **fio_fiod_cmu_dark_channel_set**(3fio), **fio_fiod_cmu_dark_channel_get**(3fio)

NAME

fio_fiod_deregister – Deregister a FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_deregister( FIO_APP_HANLE      app_handle,
                         FIO_DEV_HANDLE     dev_handle )
```

DESCRIPTION

This function is used to deregister access to a FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call.

This function cleans up, releases any memory associated with this FIOD and closes all services that may have been opened and in use by the FIOD for this application program.

If **fio_fiod_register**(3fio) has been called multiple times for the same FIOD, only one call to **fio_fiod_deregister**(3fio) is required to clean up this FIOD.

dev_handle is invalidated and may no longer be used.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

When an application program deregisters for access to a FIOD, the FIO API disables the FIOD (as if **fio_fiod_disable**(3fio) had been called), relinquish all output points for that device (as if **fio_fiod_outputs_reservation_set**(3fio) had been called with all values set to 0) and sets all application program settable states to their default values (see individual **fio**(3fio) MAN pages).

RESTRICTIONS

None

SEE ALSO

fio_deregister(3fio), **fio_fiod_register**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_query_fiod**(3fio)

NAME

fio_fiod_disable – Disable communication to a FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_disable( FIO_APP_HANLE          app_handle,
                      FIO_DEV_HANDLE         dev_handle )
```

DESCRIPTION

This function is used to disable access to a FIOD for this application program.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call.

dev_handle is invalidated and may no longer be used.

This function disables communications to a FIOD for this application program. If communications are enabled by other application programs to the indicated FIOD, no further work is done. If this is the only remaining application program requesting communications to the indicated FIOD, then communications are stopped by the FIO API to the indicated FIOD.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

When a FIOD is disabled, any output points which have been reserved by this application program are set to off.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio)

NAME

fio_fiod_enable – Enable communicates to a FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_enable( FIO_APP_HANLE          app_handle,
                     FIO_DEV_HANDLE             dev_handle )
```

DESCRIPTION

This function is used to enable access to a FIOD for this application program.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call.

The FIO API does not open any serial communications port or initiate communications to any FIOD unless explicitly commanded to do so by an application program using this function.

When this function is called, the FIO API initiates scheduled communications between the FIO API and the specified FIOD if not already active. If communications to the indicated FIOD are already in process because of another application program enabling them, no further work is done. If communications have not been enabled, the FIO API will initiate scheduled communications with the indicated FIOD.

All default frequency request frames AND any additional request frames as indicated by calling **fio_fiod_frame_schedule_set**(3fio) will be sent to the indicated FIOD in request frame sort order.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The FIO API resets all Module Status bits using the Request Module Status frame when a **FIO332**, **FIOTS1**, **FIOTS2,** or **SIU** device is first enabled. If the Module Status bits indicate hardware reset, communication loss, or watchdog reset, then the FIO API clears those bits, resets the input point filter values and reconfigures transition reporting.

Anytime a response to a Request Module Status frame has Module Status bits indicating hardware reset, communications loss, or watchdog reset, then the FIO API clears those bits, resets the input point filter values and reconfigures transition reporting.

If an application program attempts to enable a device using **fio_fiod_enable**(3fio) that has been disabled due to an API Health Monitor fault condition, then the enable operation returns an error and the FIOD remains disabled.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_disable**(3fio), **fio_query_fiod**(3fio)

NAME

fio_fiod_frame_notify_deregister – Deregister a notification request for when a response frame is received

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_frame_notify_deregister( FIO_APP_HANDLE     app_handle,
                                      FIO_DEV_HANDLE     dev_handle,
                                      unsigned int       rx_frame )
```

DESCRIPTION

This function is used to deregister a notification request for when a command frame is acknowledged (response frame is received by the FIO API) or when an error occurs.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *rx_frame* is a valid response frame number / type for which notification has been registered, using the **fio_fiod_frame_notify_register**(3fio) call. Valid response frame numbers/types are in the range 128 - 255.

Notification is generated to the application program utilizing the **FIO_SIGIO** (**SIGRTMIN** + 4) real-time signal. The application program using this service must establish a **FIO_SIGIO** handler, prior to calling this function. The default handling for a **FIO_SIGIO** real-time signal is to terminate the process. When the indicated *rx_frame* is received or declared in error by the FIO API, the FIO API will generate a **FIO_SIGIO** real-time signal to the waiting application program. The application program must then perform a **fio_query_frame_notify_status**(3fio) call to discover why a **FIO_SIGIO** real-time signal was generated. See **fio_query_frame_notify_status**(3fio) for further details. The raw data of the most recently received response frame, as indicated by *rx_frame*, may be retrieved utilizing **fio_fiod_frame_read**(3fio). Real-time **FIO_SIGIO** signals are queued to guarantee that all desired notifications are received.

An application program may register notifications for multiple response frames.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.
EACCESS	The Frame Notification Service is not currently enabled.

NOTES

The **FIO_SIGIO** real-time signal is generated by the FIO API when the indicated *rx_frame* is received for the indicated FIOD or an error has been declared. It is not considered an error to register notification for the same *rx_frame* multiple times. The values of the last call to **fio_fiod_frame_notify_register**(3fio) will be used.

The notification can be set for a one time occurrence or continuous occurrence, based upon the *notify* value passed. If **FIO_NOTIFY_ALWAYS** is passed, the FIO API automatically resets the notification process for the notification of *rx_frame*. If **FIO_NOTIFY_ONCE** is passed, the FIO API automatically disables future notification (as if **fio_fiod_frame_notify_deregister**(3fio) has been called) of the *rx_frame*.

Real-time **FIO_SIGIO** signals are queued to guarantee that all desired notifications are received.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_frame_schedule_get**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_read**(3fio), **fio_fiod_frame_notify_register**(3fio), **fio_query_frame_notify_status**(3fio)

NAME

fio_fiod_frame_notify_register – Register a notification request for when a response frame is received

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_frame_notify_register( FIO_APP_HANDLE     app_handle,
                                    FIO_DEV_HANDLE     dev_handle,
                                    unsigned int       rx_frame,
                                    FIO_NOTIFY         notify )
```

DESCRIPTION

This function is used to register a notification request for when a command frame is acknowledged (response frame is received by the FIO API) or when an error occurs.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *rx_frame* is a valid response frame number/type for which notification is to be registered. Valid response frame numbers/types are in the range 128 - 255. *notify* is an indication as to the frequency of notification; **FIO_NOTIFY_ONCE** and **FIO_NOTIFY_ALWAYS** may be specified to set notification for one occurrence or for all occurrences, respectively.

Notification is generated to the application program utilizing the **FIO_SIGIO** (**SIGRTMIN** + 4) real-time signal. The application program using this service must establish a **FIO_SIGIO** handler, prior to calling this function. The default handling for a **FIO_SIGIO** real-time signal is to terminate the process. When the indicated *rx_frame* is received or declared in error by the FIO API, the FIO API will generate a **FIO_SIGIO** real-time signal to the waiting application program. The application program must then perform a **fio_query_frame_notify_status**(3fio) call to discover why a **FIO_SIGIO** real-time signal was generated. See **fio_query_frame_notify_status**(3fio) for further details. The raw data of the most recently received response frame, as indicated by *rx_frame*, may be retrieved utilizing **fio_fiod_frame_read**(3fio). Real-time **FIO_SIGIO** signals are queued to guarantee that all desired notifications are received.

An application program may register notifications for multiple response frames.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The **FIO_SIGIO** real-time signal is generated by the FIO API when the indicated *rx_frame* is received for the indicated FIOD or an error has been declared. It is not considered an error to register notification for the same *rx_frame* multiple times. The *notify* value of the last call to **fio_fiod_frame_notify_register**(3fio) will be used.

The notification can be set for a one time occurrence or continuous occurrence, based upon the *notify* value passed. If **FIO_NOTIFY_ALWAYS** is passed, the FIO API automatically resets the notification process for the notification of *rx_frame*. If **FIO_NOTIFY_ONCE** is passed, the FIO API automatically disables future notification (as if **fio_fiod_frame_notify_deregister**(3fio) has been called) of the *rx_frame* response frame.

Real-time **FIO_SIGIO** signals are queued to guarantee that all desired notifications are received.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_frame_schedule_get**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_read**(3fio), **fio_fiod_frame_notify_deregister**(3fio), **fio_query_frame_notify_status**(3fio)

NAME

fio_fiod_frame_read – Read the raw data of the most recent response frame

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_frame_read( FIO_APP_HANDLE      app_handle,
                         FIO_DEV_HANDLE      dev_handle,
                         unsigned int        rx_frame,
                         unsigned int        *seq_number,
                         unsigned char       *buf,
                         unsigned int        count )
```

DESCRIPTION

This function is used to read the raw data from the most recent response frame for the *rx_frame* requested.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *rx_frame* is a valid response frame number / type for which data is to be returned. Valid response frame numbers / types are in the range 128 - 255. *seq_number* is a pointer to an unsigned int that will contain a relative sequence number upon successful return. *seq_number* may be **NULL**. In which case, the *seq_number* is not returned. *Buf* is a pointer to an unsigned character array, of *count* bytes, that will contain the raw response frame data upon successful completion. *count* is a count of the number of bytes found in *frame*.

The raw data of the most recently received response frame, as indicated by *rx_frame*, is returned in *buf* upon successful completion.

RETURN VALUES

Upon successful completion, the number of bytes read is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

It is possible that a value of 0 is returned. This is not considered an error. It is possible that the response frame requested has not yet been received or was in error. Use **fio_fiod_frame_notify_register**(3fio) to enable application program notification of the arrival of a specific response frame.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

Less than the requested number of bytes can and will be returned if the raw data is smaller than the *buf* supplied. The number of bytes read is returned.

An incomplete read of the raw response frame data can and will occur if the *buf* passed is too small to hold the raw response frame data. The number of bytes read is returned.

seq_number is unique to each individual response frame number/type and is incremented for that response frame number/type each time a response frame of that unique number/type is received. The FIO API performs no roll-over processing on the *seq_number*. When the maximum value of an unsigned int is reached, the next *seq_number* will be 0. The application program must handle accordingly.

It is possible that a response frame size returned by **fio_fiod_frame_size**(3fio) DOES NOT match the response size returned from a subsequent **fio_fiod_frame_read**(3fio) call. This will happen if the size of the response frame indicated can be variable length AND a new response frame is received by the FIOD between the time after a call to **fio_fiod_frame_size**(3fio) and a subsequent call to **fio_fiod_frame_read**(3fio). Application developers must be aware of this possibility. This case may be identified by the fact that the *seq_number* returned by these 2 calls is different. To be safe, applications should always use the maximum size of the response frame as the size of its buffer.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_frame_schedule_get**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_notify_register**(3fio), **fio_query_frame_notify_status**(3fio)

NAME

fio_fiod_frame_schedule_get – Get the current active request frame schedule for a FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_frame_schedule_get( FIO_APP_HANDLE app_handle,
                                 FIO_DEV_HANDLE     dev_handle,
                                 FIO_VIEW           view,
                                 FIO_FRAME_SCHD     *frame_schd,
                                 unsigned int       count )
```

DESCRIPTION

This function is used to get the current frequency at which the indicated request frame(s) is to be sent to the indicated FIOD, periodically.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *view* is an indication as to the view of request frame frequency data to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *frame_schd* is a pointer to an array of **FIO_FRAME_SCHD** structures. *count* is a count of the number of *frame_schd* elements passed.

FIO_FRAME_SCHD is defined as:

```
struct
{
        unsigned int        req_frame;      /* Request Frame # */
        FIO_HZ              frequency;      /* Frequency in Hz of this frame */
} fio_frame_schd;
typedef struct fio_frame_schd FIO_FRAME_SCHD;
```

req_frame must be set to a valid request frame number, supported by the FIO API. *frequency* is returned with the current frequency used for the indicated request frame. A value of **FIO_HZ_0** (0) is used to indicate that the request frame is disabled. Only the indicated request frame frequency values are returned. If an application program desires to know frequencies for all possible request frames, a fully populated *frame_schd* list must be passed. Valid request frames numbers are in the range of 0 – 127; but are limited to those specified by the requirements. If an application program passes a *req_frame* number that is not supported by the FIO API, this is not an error, but **FIO_HZ_0** (0) will be returned for all such request frames.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

Upon successful return, the *frame_schd* array will contain the actual frequencies currently being utilized by the FIO API for the *view* indicated.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

None

RESTRICTIONS

Only valid supported request frame frequencies will be returned. An application program can request the frequency of a non-supported request frame, but the returned frequency will always be **FIO_HZ_0** (0).

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_read**(3fio), **fio_fiod_frame_schedule_get**(3fio)

NAME

fio_fiod_frame_schedule_set – Schedule a request frame to be sent periodically to a FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_frame_schedule_set( FIO_APP_HANDLE      app_handle,
                                 FIO_DEV_HANDLE      dev_handle,
                                 FIO_FRAME_SCHD      *frame_schd,
                                 unsigned int        count )
```

DESCRIPTION

This function is used to set scheduled request frame frequencies for request frames for which the indicated request frame(s) is/are to be sent to the indicated FIOD periodically for a registered FIOD. This function is used to modify default request frame frequencies; whether the default is **FIO_HZ_0** (0) or another value.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *count* is a count of the number of *frame_schd* elements passed. *frame_schd* is a pointer to an array of **FIO_FRAME_SCHD** structures.

FIO_FRAME_SCHD is defined as:

```
struct
{
        unsigned int        req_frame;      /* Request Frame # */
        FIO_HZ              frequency;      /* Frequency in Hz of this frame */
} fio_frame_schd;
typedef struct fio_frame_schd FIO_FRAME_SCHD;
```

req_frame must be set to a valid request frame number, supported by the FIO API; as defined by the requirements. *frequency* must be set to a frequency supported by the FIO API. A value of **FIO_HZ_0** (0) is used to disable the periodic sending of a request frame.

Request frame frequency setting is atomic; all or nothing. If any value in the *frame_schd* array is found to be invalid, the entire operation will fail.

Request frame frequencies that are not indicated in the *frame_schd* array are not touched; frequencies set by other application programs are maintained and utilized. The request frame frequency utilized by the FIO API is the highest frequency requested by all application programs registered for that FIOD. For a request frame to be disabled to a FIOD, all registered application programs must set the request frame frequency to **FIO_HZ_0** (0).

A request frame may be sent one time utilizing the frequency **FIO_HZ_ONCE**. However, in this case, if the request frame currently has a higher frequency, this higher frequency will be utilized.

Once a request frame frequency has been established AND communications with the indicated FIOD have been established, the indicated request frame will commence being sent at the indicated frequency to the indicated FIOD.

If an application program deregisters with a FIOD, the request frame frequency for all request frames associated with that application program are set to **FIO_HZ_0** (0), and the next highest frequencies for those request frames are utilized by the FIO API.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

Upon successful return, the *frame_schd* array will contain the actual request frame frequencies currently being utilized by the FIO API system view.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The ATC Controller Standard, Section 8, specifies the frames for communication with FIODs for Model 332 Cabinets, NEMA TS 1 and TS 2 Type 2 Cabinets and ITS Cabinets. The FIO API supports a subset of these frames at the scheduled frame frequencies as shown in Table 3 of the SRS.

The NEMA TS 2 Standard, Section 3.3, specifies the frames for communication with FIODs for NEMA TS 2 Type 1 Cabinets. The FIO API supports a subset of these frames at the scheduled frame frequencies as shown in Table 4 of the SRS.

The timing for the command/response cycle of the frames is defined by the “Handshaking” algorithm in Section 3.3.1.5.3 of the NEMA TS 2 Standard. Although the timing for the command/response cycle of the frames is defined for the FIO API by the NEMA TS 2 Standard, it applies to all frame types regardless of the cabinet standard.

A frame intended to be sent one time will be sent after the next set of scheduled frames has completed. Generally, the FIO API orders the frames sent to a FIOD based on their frame type, grouping frames of like frequencies together.

The FIO API resets all Module Status bits using the Request Module Status frame when a **FIO332**, **FIOTS1**, **FIOTS2,** or **SIU** device is first enabled. If the Module Status bits indicate hardware reset, communication loss, or watchdog reset, then the FIO API clears those bits, resets the input point filter values and reconfigures transition reporting.

Anytime a response to a Request Module Status frame has Module Status bits indicating hardware reset, communications loss, or watchdog reset, then the FIO API clears those bits, resets the input point filter values and reconfigures transition reporting.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_read**(3fio), **fio_fiod_frame_schedule_get**(3fio)

NAME

fio_fiod_frame_size – Retrieve the size of a response frame

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_frame_size( FIO_APP_HANDLE      app_handle,
                         FIO_DEV_HANDLE      dev_handle,
                         unsigned int        rx_frame,
                         unsigned int        *seq_number )
```

DESCRIPTION

This function is used to retrieve the size of the most recent response frame for the *rx_frame* requested.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *rx_frame* is a valid response frame number/type for which its size in bytes is to be returned. Valid response frame numbers/types are in the range 128 - 255. *seq_number* is a pointer to an unsigned int that will contain a relative sequence number upon successful return. *seq_number* may be **NULL**. In which case, the *seq_number* is not returned.

The size in bytes of the most recently received response frame, as indicated by *rx_frame*, is returned upon successful completion.

RETURN VALUES

Upon successful completion, the number of bytes available for the response frame indicated is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

It is possible that a value of 0 is returned. This is not considered an error. It is possible that the response frame requested has not yet been received or was in error. Use **fio_fiod_frame_notify_register**(3fio) to enable application program notification of the arrival of a specific response frame.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

It is possible that a response frame size returned by this function DOES NOT match the response size returned from a subsequent **fio_fiod_frame_read**(3fio) call. This will happen if the size of the response frame indicated can be variable length AND a new response frame is received by the FIOD between the time after a call to **fio_fiod_frame_size**(3fio) and a subsequent call to **fio_fiod_frame_read**(3fio). Application developers must be aware of this possibility. This case may be identified by the fact that the *seq_number* returned by these 2 calls is different. To be safe, applications should always use the maximum size of the response frame as the size of its buffer.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_frame_schedule_get**(3fio), **fio_fiod_frame_read**(3fio), **fio_fiod_frame_notify_register**(3fio), **fio_query_frame_notify_status**(3fio)

NAME

fio_fiod_inputs_filter_get – Get the leading and trailing edge filter for a configurable FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_inputs_filter_get( FIO_APP_HANDLE          app_handle,
                                FIO_DEV_HANDLE          dev_handle,
                                FIO_VIEW                view,
                                FIO_INPUT_FILTER        *input_filter,
                                unsigned int            count )
```

DESCRIPTION

This function is used to get the current leading and trailing edge filter values on a per input basis for all FIODs that support configurable filtered inputs.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *view* is an indication as to the view of input filter value data to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *input_filter* is a pointer to an array of **FIO_INPUT_FILTER** structures. *count* is a count of the number of *input_filter* elements passed.

FIO_INPUT_FILTER is defined as:

```
struct
{
        unsigned int        input_point;    /* Input Point Number */
        unsigned int        leading;        /* Leading Edge Filter Value */
        unsigned int        trailing;       /* Trailing Edge Filter Value */
} fio_input_filter;
typedef struct fio_input_filter FIO_INPUT_FILTER;
```

input_point must be set to a valid input point number. *leading* is returned with the current leading edge input filter value for the *view* indicated. *trailing* is returned with the current trailing edge input filter value for the *view* indicated. Only the indicated input filter values, for the specified *input_point*, are returned. If an application program desires to know the input filter values for all possible input points, a fully populated *input_filter* list must be passed.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

Upon successful return, the *input_filter* array will contain the actual input filter values currently being utilized by the FIO API; for the *view* indicated.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable(3fio),**
fio_fiod_inputs_get(3fio), **fio_fiod_inputs_filter_set**(3fio), **fio_fiod_frame_schedule_set**(3fio)

NAME

fio_fiod_inputs_filter_set – Set the leading and trailing edge filter for a configurable FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_inputs_filter_set( FIO_APP_HANDLE          app_handle,
                                FIO_DEV_HANDLE          dev_handle,
                                FIO_INPUT_FILTER        *input_filter,
                                unsigned int            count )
```

DESCRIPTION

This function is used to set the leading and trailing edge filter values on a per input basis for all FIODs that support configurable filtered inputs.

If multiple application programs set the filter values of an input, the shortest filter values are used. The default leading and trailing edge filter values are **FIO_FILTER_DEFAULT** (5 consecutive samples).

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *input_filter* is a pointer to an array of **FIO_INPUT_FILTER** structures. *count* is a count of the number of *input_filter* elements passed.

FIO_INPUT_FILTER is defined as:

```
struct
{
        unsigned int        input_point;    /* Input Point Number */
        unsigned int        leading;        /* Leading Edge Filter Value */
        unsigned int        trailing;       /* Trailing Edge Filter Value */
} fio_input_filter;
typedef struct fio_input_filter FIO_INPUT_FILTER;
```

input_point must be set to a valid input point number, for the indicated FIOD. *leading* is a value that indicates the sampling rate for 1 to 0 transitions. *trailing* is a value that indicates the sample rate for 0 to 1 transitions. The value **FIO_FILTER_DEFAULT** may be used to reset to the default sample rate.

Input filter setting is atomic; all or nothing. If any value in the *input_filter* array is found to be invalid, the entire operation will fail.

Input filter values that are not indicated in the *input_filter* array are not touched; filter values set by other application programs are maintained and utilized. The input filter value utilized by the FIO API is the shortest filter values requested by all application programs registered for that FIOD.

If an application program deregisters with a FIOD, the input filter values for all input filter values associated with that application program are removed and the next highest filter values are utilized by the FIO API.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

Upon successful return, the *input_filter* array will contain the actual input filter values currently being utilized by the FIO API system view.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable(3fio),** **fio_fiod_inputs_get**(3fio), **fio_fiod_inputs_filter_get**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_inputs_trans_set**(3fio)

NAME

fio_fiod_inputs_get – Retrieve the current state of an FIOD input points

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_inputs_get( FIO_APP_HANDLE      app_handle,
                         FIO_DEV_HANDLE      dev_handle,
                         FIO_INPUTS_TYPE     inputs_type,
                         unsigned char       *data )
```

DESCRIPTION

This function is used to retrieve the current state of a FIOD input points. Either filtered or unfiltered (raw) input point information may be retrieved, depending upon the transmission frames that have been configured using **fio_fiod_frame_schedule_set**(3fio) and the FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *inputs_type* is an indication as to the type of input point data to return; **FIO_INPUTS_RAW** and **FIO_INPUTS_FILTERED** may be specified. For FIODs that only support one type of input, *inputs_type* returns the same data regardless of its value. *data* is a pointer to an unsigned character buffer that will contain a bit array of the current state of the input points; upon successful completion. This buffer must be at least **FIO_INPUT_POINTS_BYTES** in length; no checking is performed by this function to ensure this.

If the appropriate frame has been configured, *data* will be filled with the bit array of the input points. If the appropriate frame has not been configured, *data* will be filled with an array of all 0 bits.

The macro **FIO_BIT_TEST**(*addr*, *num*) may be used to test for a bit being set in the resulting bit array; where *addr* is the address of the bit array (*data*) and *num* is the bit number of the bit to be tested. No boundary checking is performed by this macro; the user must perform all boundary checking.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

If multiple application programs have registered for the same FIOD, the FIO API provides shared read access to the input point states for all application programs which have registered that device.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_outputs_set**(3fio)

NAME

fio_fiod_inputs_trans_get – Get the current transition buffer input points monitoring state

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_inputs_trans_get( FIO_APP_HANDLE      app_handle,
                               FIO_DEV_HANDLE      dev_handle,
                               FIO_VIEW            view,
                               unsigned char       *data )
```

DESCRIPTION

This function is provided to allow application programs to view input points that enabled for transition monitoring.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *view* is an indication as to the view of input point transition monitoring data to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *data* is a pointer to an unsigned character buffer that will contain a bit array of the input point transitions for this FIOD upon successful completion. Bits that are monitored will be set to 1 in this bit array. Bits that are ignored (not monitored) will be set to 0 in this bit array. This buffer must be at least **FIO_INPUT_POINTS_BYTES** in length; no checking is performed by the function to ensure this. This bit array is a "positive" mask; the states of bits in this array are the current *view* of the state (monitored / not monitored) of input point transitions for this application program (**FIO_VIEW_APP**) or system (**FIO_VIEW_SYSTEM**)

By default, transition monitoring for all input points is disabled.

The macro **FIO_BIT_TEST**(*addr*, *num*) may be used to test for a bit being set in the resulting bit array; where *addr* is the address of the bit array (*data*) and *num* is the bit number of the bit to be tested. No boundary checking is performed by this macro; the user must perform all boundary checking.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

The FIO API has the ability to collect and buffer the transition buffer information for each registered FIOD used for input.

Transition buffer frames, for the indicated FIOD, must be scheduled using **fio_fiod_frame_schedule_set**(3fio) in order to receive input point transition information.

The **fio_fiod_inputs_trans_set**(3fio) function utilizes the "positive" mask for enabling/disabling monitoring input point transitions.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable(3fio),**
fio_fiod_inputs_get(3fio), **fio_fiod_inputs_filter_set**(3fio), **fio_fiod_inputs_filter_get**(3fio),
fio_fiod_frame_schedule_set(3fio), **fio_fiod_inputs_trans_set**(3fio),
fio_fiod_inputs_trans_read(3fio)

NAME

fio_fiod_inputs_trans_read – Read the current input points transition buffer data

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_inputs_trans_read( FIO_APP_HANDLE          app_handle,
                                FIO_DEV_HANDLE          dev_handle,
                                FIO_TRANS_STATUS        *status
                                FIO_TRANS_BUFFER        *trans_buf,
                                unsigned int            count )
```

DESCRIPTION

This function allows an application program to access the FIO API transition buffer information asynchronously (i.e. read the transition entries from the FIO API buffer independent of any FIOD communications).

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *status* is a pointer to a **FIO_TRANS_STATUS** value. Upon successful completion this value will indicate the transition buffer status: **FIO_TRANS_SUCCESS** indicates successful transition buffer handling, **FIO_TRANS_FIOD_OVERRUN** indicates the FIOD transition buffer experienced an overrun condition and **FIO_TRANS_APP_OVERRUN** indicates the application program transition buffer experienced as overrun condition. *trans_buf* is a pointer to an array of **FIO_TRANS_BUFFER** structures. *count* is a count of the number of *trans_buf* elements passed.

FIO_TRANS_BUFFER is defined as:

```
struct
{
        unsigned int    input_point : 7;  /* Input number */
        unsigned int    state : 1;        /* Input state 0 = off, 1 = on */
        unsigned int    timestamp : 16; /* 16 bit timestamp for FIOD */
} fio_trans_buffer;
typedef struct fio_trans_buffer FIO_TRANS_BUFFER;
```

input_point will be set to a valid input point number. Only input transitions for the input points enabled via a successful **fio_fiod_inputs_trans_set**(3fio) call will be returned. *state* is returned with the current state of the input point, at the *timestamp* indicated. *timestamp* is returned with the 16 bit timestamp returned from the FIOD.

RETURN VALUES

Upon successful completion, the count of the number of filled *trans_buf* structures is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

The FIO API has the ability to collect and buffer the transition buffer information for each registered FIOD used for input.

When the FIO API reads the transition buffer of a fiod, it reads the entire transition buffer.

The FIO API buffers the transition data on a per application program basis with the capability of storing 1024 transition entries in a FIFO fashion.

When an application program reads a transition entry from an FIO API transition buffer, that transition entry is cleared for that application program only, without affecting the FIO API transition buffers for other application programs.

If the transition buffer in the FIOD overruns before information can be copied to the FIO API transition buffer information, the FIO API indicates that a device overrun condition has occurred in the transition buffer for that FIOD by returning a *status* of **FIO_TRANS_FIOD_OVERRUN**. If the transition buffer of the FIO API overruns before the information is retrieved by the application program, the FIO API indicates that a FIO API overrun condition has occurred by returning a *status* of **FIO_TRANS_APP_OVERRUN**. Both overrun conditions are announced one time to the application program and imply that data was lost.

Transition buffer frames, for the indicated FIOD, must be scheduled using **fio_fiod_frame_schedule_set**(3fio) in order to receive input point transition information.

The **fio_fiod_inputs_trans_set**(3fio) function utilizes a “positive” mask for enabling/disabling monitoring input point transitions.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_inputs_filter_set**(3fio), **fio_fiod_inputs_filter_get**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_inputs_trans_set**(3fio), **fio_fiod_inputs_trans_get**(3fio)

NAME

fio_fiod_inputs_trans_set – Set the current transition buffer input points desired

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_inputs_trans_set( FIO_APP_HANDLE      app_handle,
                               FIO_DEV_HANDLE      dev_handle,
                               unsigned char       *data )
```

DESCRIPTION

This function is provided to allow application programs to enable or disable transition monitoring of selected input points.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *Data* is a pointer to an unsigned character buffer that contains a bit array of the desired input point transitions for this FIOD for this application program. Bits that are to be monitored are set to 1 in this bit array. Bits that are to be ignored (not monitored) are set to 0 in this bit array. This buffer must be at least **FIO_INPUT_POINTS_BYTES** in length; no checking is performed by the function to ensure this. This bit array is a “positive” mask; the states of bits in this array are this application programs current view of the state (monitored/not monitored) of input point transitions from its perspective.

By default, transition monitoring for all input points is disabled.

If an application program enables an input point for transition monitoring and that input point is already in the enabled state, it is not considered an error. If an application program disables an input point for transition monitoring and that input point is already in the disabled state, it is not considered an error. The bit array is a “positive” mask.

The macros **FIO_BIT_SET**(*addr*, *num*) and **FIO_BIT_CLEAR**(*addr*, *num*) may be used to set and clear a bit in a bit array; where *addr* is the address of the bit array (*data*) and *num* is the bit number of the bit to be set/cleared. No boundary checking is performed by these macros; the user must perform all boundary checking.

The macro **FIO_BITS_CLEAR**(*addr*, *size*) may be used to initialize a bit array to the 0 state; where *addr* is the address of the bit array (*data*) and *size* is the size of the bit array in bytes (**FIO_INPUT_POINTS_BYTES**).

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The FIO API has the ability to collect and buffer the transition buffer information for each registered FIOD used for input.

Transition buffer frames, for the indicated FIOD, must be scheduled using **fio_fiod_frame_schedule_set**(3fio) in order to receive input point transition information.

The **fio_fiod_inputs_trans_get**(3fio) function returns the “positive” mask for either this application program’s or the current system input point monitoring state.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable(3fio), fio_fiod_inputs_get**(3fio), **fio_fiod_inputs_filter_set**(3fio), **fio_fiod_inputs_filter_get**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_inputs_trans_get**(3fio), **fio_fiod_inputs_trans_read**(3fio)

NAME

fio_fiod_mmu_flash_bit_get – Get the **FIOMMU** Load Switch Flash Bit state

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_mmu_flash_bit_get( FIO_APP_HANDLE      app_handle,
                                FIO_DEV_HANDLE      dev_handle,
                                FIO_VIEW            view,
                                FIO_MMU_FLASH_BIT   *fb )
```

DESCRIPTION

This function provides a method to get the state of the Load Switch Flash bit of a registered **FIOMMU** FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOMMU** FIOD. *view* is an indication as to the view of *fb* to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified to view application program specific data or system data, respectively. *fb* is a pointer to **FIO_MMU_FLASH_BIT** that will contain the **FIOMMU** Load Switch Flash Bit value upon successful completion. Valid values are **FIO_MMU_FLASH_BIT_OFF** and **FIO_MMU_FLASH_BIT_ON**.

The **FIO_VIEW_SYSTEM** *view* is the current Load Switch Flash Bit value being utilized by the system for this **FIOMMU** FIOD. The default *fb* is **FIO_MMU_FLASH_BIT_OFF**.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The last Load Switch Flash Bit value set by all application programs is used by the system.

RESTRICTIONS

dev_handle must refer to a **FIOMMU** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_mmu_flash_bit_set**(3fio)

NAME

fio_fiod_mmu_flash_bit_set – Select a **FIOMMU** Load Switch Flash Bit state

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_mmu_flash_bit_set( FIO_APP_HANDLE      app_handle,
                                FIO_DEV_HANDLE      dev_handle,
                                FIO_MMU_FLASH_BIT   fb )
```

DESCRIPTION

This function provides a method to set the state of the Load Switch Flash bit of a registered **FIOMMU** FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOMMU** FIOD. *fb* is a valid **FIO_MMU_FLASH_BIT** that contains the **FIOMMU** Load Switch Flash Bit value. Valid values are **FIO_MMU_FLASH_BIT_OFF** and **FIO_MMU_FLASH_BIT_ON**.

The last application program to set the Load Switch Flash Bit is used by the system. The default Load Switch Flash Bit is **FIO_MMU_FLASH_BIT_OFF**.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

If multiple application programs attempt to set the state of the Load Switch Flash bit, the FIO API uses the most recent state.

RESTRICTIONS

dev_handle must refer to a **FIOMMU** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_mmu_flash_bit_get**(3fio)

NAME

fio_fiod_outputs_get – Retrieve the current state of a FIOD output points

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_outputs_get( FIO_APP_HANDLE      app_handle,
                          FIO_DEV_HANDLE      dev_handle,
                          FIO_VIEW            view,
                          unsigned char       *ls_plus,
                          unsigned char       *ls_minus )
```

DESCRIPTION

This function is used to retrieve the current state of a FIOD output points. Either the current application programs view or the current system view of output point information may be retrieved.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *view* is an indication as to the view of output point data to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *ls_plus* is a pointer to an unsigned character buffer that will contain a bit array of the current state of the “Load Switch Plus” output points; upon successful completion. *ls_minus* is a pointer to an unsigned character buffer that will contain a bit array of the current state of the “Load Switch Minus” output points upon successful completion. These buffers must each be at least **FIO_OUTPUT_POINTS_BYTES** in length; no checking is performed by the function to ensure this.

The macro **FIO_BIT_TEST**(*addr*, *num*) may be used to test for a bit being set in the resulting bit arrays; where *addr* is the address of the bit array (*ls_plus* or *ls_minus*) and *num* is the bit number of the bit to be tested. No boundary checking is performed by this macro; the user must perform all boundary checking.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

If multiple application programs have registered for the same FIOD, the FIO API provides shared read access to the output point states for all application programs which have registered that device.

When the state of an output point is read, the FIO API returns the current state of that output point within the FIO API.

Exclusive reservation of an output point for write access by one application program does not preclude other application programs from reading the state of the output point.

Output points that may be queried are specific per unique FIOD.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_outputs_reservation_get**(3fio)

NAME

fio_fiod_outputs_reservation_get – Get the reservation state of an FIOD output point

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_outputs_reservation_get( FIO_APP_HANDLE        app_handle,
                                      FIO_DEV_HANDLE        dev_handle,
                                      FIO_VIEW              view,
                                      unsigned char         *data )
```

DESCRIPTION

This function is used to query the current reservation state of a FIOD output points. Either the current application program's view or the current system view of output point reservation information may be retrieved.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *view* is an indication as to the view of output point reservation data to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified to view application program specific data or system data, respectively. *data* is a pointer to an unsigned character buffer that will contain a bit array of the current reservation state of the output points; upon successful completion. This buffer must be at least **FIO_OUTPUT_POINTS_BYTES** in length; no checking is performed by the function to ensure this. A 1 bit indicates that the corresponding output point is reserved, a 0 bit indicates that the corresponding output point is relinquished, not reserved. This bit array is a "positive" mask. The construction of this bit array is directly useable by **fio_fiod_outputs_reservation_set**(3fio).

The macro **FIO_BIT_TEST**(*addr*, *num*) may be used to test for a bit being set in the resulting bit array, where *addr* is the address of the bit array (*data*) and *num* is the bit number of the bit to be tested. No boundary checking is performed by this macro; the user must perform all boundary checking.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL One or more arguments are invalid.

NOTES

Output points that may be reserved are specific per unique FIOD.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_ena**ble(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_outputs_reservation_set**(3fio)

NAME

fio_fiod_outputs_reservation_set – Set the reservation state of a FIOD output point

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_outputs_reservation_set( FIO_APP_HANDLE        app_handle,
                                      FIO_DEV_HANDLE        dev_handle,
                                      unsigned char         *data )
```

DESCRIPTION

This function provides a method for application programs to reserve/relinquish exclusive (single application program) “write access” to individual output points of a FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *Data* is a pointer to an unsigned character buffer that contains a bit array of the desired reservation states of the output points for this FIOD, for this application program. Bits that are to be reserved are set to 1 in this bit array. Bits that are to be relinquished (not reserved) are set to 0 in this bit array. This buffer must be at least **FIO_OUTPUT_POINTS_BYTES** in length; no checking is performed by the function to ensure this. This bit array is a “positive” mask; the states of bits in this array are this application programs current view of the state (reserved / relinquished) of output points from its perspective.

If an application program attempts to reserve a point that has already been reserved by that application program, it is not considered an error. If an application program relinquishes a point that is already in the relinquished state for that application program, it is not considered an error. The bit array is a “positive” mask.

If a point in a group of not already reserved points cannot be reserved, the reservation attempt fails for the entire new group of points; **ENOTTTY** is returned in *errno*. The reservation state of previously reserved points is maintained; both reserved and relinquished state of points. The reserve / relinquish operation is atomic; all or nothing.

The macros **FIO_BIT_SET**(*addr*, *num*) and **FIO_BIT_CLEAR**(*addr*, *num*) may be used to set and clear a bit in a bit array; where *addr* is the address of the bit array (*data*) and *num* is the bit number of the bit to be set/cleared. No boundary checking is performed by these macros; the user must perform all boundary checking.

The macro **FIO_BITS_CLEAR**(*addr*, *size*) may be used to initialize a bit array to the 0 state; where *addr* is the address of the bit array (*data*) and *size* is the size of the bit array in bytes (**FIO_OUTPUT_POINTS_BYTES**).

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOTTY	One or more of the requested output points is already reserved by another application program.

NOTES

The FIO API allows only one application program to reserve write access to any individual output point. The FIO API allows multiple application programs to reserve different output points on a single FIOD. The FIO API provides error codes so that the application program can determine if the reservation action was successful or if there was a conflict with another application program.

Relinquishing a reserved output point or channel clears the channel map for the applicable output points.

The FIO API makes output point reservations on a “first come first served basis.” Setting the *data* bit array to all 0’s prior to this call will result in all reservations being relinquished.

The **fio_fiod_outputs_reservation_get**(3fio) function returns the “positive” mask for either this application program’s or the current system reservations.

The FIO API retains ownership of the Watchdog, Fault Monitor and Volt Monitor output points and does not allow application programs to reserve these output points directly via this function.

Output points that may be reserved are specific per unique FIOD.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_get**(3fio), **fio_fiod_outputs_reservation_get**(3fio), **fio_fiod_wd_reservation_set**(3fio)

NAME

fio_fiod_outputs_set – Set the current state of a FIOD output points

SYNOPSIS

```
#include      <fio.h>

int fio_fiod_outputs_set( FIO_APP_HANDLE      app_handle,
                          FIO_DEV_HANDLE      dev_handle,
                          unsigned char       *ls_plus,
                          unsigned char       *ls_minus )
```

DESCRIPTION

This function is used to set the current state of a FIOD output points.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *ls_plus* is a pointer to an unsigned character buffer that contains a bit array of the current state of the "Load Switch Plus" output points to be set. *ls_minus* is a pointer to an unsigned character buffer that contains a bit array of the current state of the "Load Switch Minus" output points to be set. These buffers must each be at least **FIO_OUTPUT_POINTS_BYTES** in length; no checking is performed by the function to ensure this.

An application program is able to set the state of an output point if it has registered the associated FIOD and reserved exclusive write access to the output point. An attempt to turn on or off a non-reserved output point is ignored. Only output points that are reserved by this application program are affected by this call. Non-NEMA devices set their outputs using arrays of Data and Control instead of Load Switch +/-. The FIO API makes the translation accordingly. For FIODs that do not support plus/minus or data/control, the *ls_plus* and *ls_minus* arrays are OR'ed together to get the resulting bit that is written to the device.

The macros **FIO_BIT_SET**(*addr*, *num*) and **FIO_BIT_CLEAR**(*addr*, *num*) may be used to set and clear a bit in a bit array; where *addr* is the address of the bit array (*ls_plus* or *ls_minus*) and *num* is the bit number of the bit to be set/cleared. No boundary checking is performed by these macros; the user must perform all boundary checking.

The macro **FIO_BITS_CLEAR**(*addr*, *size*) may be used to initialize a bit array to the 0 state; where *addr* is the address of the bit array (*ls_plus* or *ls_minus*) and *size* is the size of the bit array in bytes (**FIO_OUTPUT_POINTS_BYTES**).

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

Only reserved output points are affected by a call to this function.

The FIO API updates the mapped channel and color on a **FIOMMU** or **FIOCMU** device based on the value of the associated output point. For a **FIOMMU** the output point plus and minus values are directly mapped to **FIOMMU** plus/minus. For a **FIOCMU**, the channel is set to off if, and only if, both plus and minus are set to off.

Non-NEMA devices set their outputs using arrays of Control and Data instead of Load Switch +/-. The FIO API makes the translation accordingly.

Output points that may be updated are specific per unique FIOD.

The FIO API uses channel mappings, set using **fio_fiod_channel_map_set**(3fio), to set the contents of **FIOMMU** Frame 0 and **FIOCMU** Frames 61 and 67, from the values set using this function.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_enable**(3fio), **fio_fiod_inputs_get**(3fio), **fio_fiod_outputs_set**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_outputs_reservation_get**(3fio), **fio_fiod_channel_map_set**(3fio)

NAME

fio_fiod_register – Register with the FIO API to access services for a FIOD

SYNOPSIS

```
#include        <fio.h>

FIO_DEV_HANDLE fio_fiod_register( FIO_APP_HANDLE    app_handle,
                                  FIO_PORT          port,
                                  FIO_DEVICE_TYPE   dev )
```

DESCRIPTION

This function is used to gain access to FIOD services provided by the FIO API.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful call to **fio_register**(3fio). *Port* is a **FIO_PORT** that is a valid serial port to which the FIOD is connected. *Dev* is a **FIO_DEVICE_TYPE** that indicates the type of FIOD that is being accessed.

If the same FIOD is registered multiple times, the same **FIO_DEV_HANDLE** is returned. Only one call to **fio_fiod_deregister**(3fio) is needed to clean up.

RETURN VALUES

Upon successful completion a valid **FIO_DEV_HANDLE** is returned. This handle must be used in all subsequent **fio_fiod**(3fio) function calls.

A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.
ENODEV	The indicated port is not open.
ECONNREFUSED	FIOD being registered is not compatible with other FIODs already connected to this port.

NOTES

The supported FIO serial communications ports are **FIO_SP3, FIO_SP5** and **FIO_SP8**. The supported communication modes on these ports are 153.6 Kbps and 614.4 Kbps SDLC. The FIO API supports communication to multiple FIODs on a single communications port provided the FIODs have compatible physical communication attributes.

The FIO API does not open any serial communications port or initiate communications to any FIOD unless explicitly commanded to do so by an application program using the **fio_fiod_enable**(3fio) function.

The FIO API supports the FIOD types listed by **FIO_DEVICE_TYPE**. The FIO API assumes that BIU and MMU FIODs operate at 153.6 Kbps and all other FIOD types operate at 614.4 Kbps. The FIO API supports a maximum of one FIOD of each type per communications port except in the case of BIUs and SIUs. The FIO API supports up to 8 Detector BIU and 8 Terminal & Facilities BIU FIODs per communications port. The FIO API supports up to 5 Input SIU, 2 14-Pack Output SIU and 4 6-Pack Output SIU FIODs per communications port. The FIO API only supports valid Output SIU combinations as defined in the ITS Cabinet Standard, Section 4.7.

Once a device has been registered on a communications port, the FIO API permits the registration of additional compatible FIODs on the same communications port and prohibits the registration of incompatible FIODs on the same communications port. If an incompatibility is detected, a negative value is returned and *errno* will be set to **ECONNREFUSED**.

The FIOD registration process does not cause the FIO API to perform any device communications, **fio_fiod_enable**(3fio) must be used to commence communications.

RESTRICTIONS

This call must be made before any other **fio_fiod**(3fio) function calls can be used.

SEE ALSO

fio_register(3fio), **fio_fiod_deregister**(3fio), **fio_fiod_enable**(3fio), **fio_query_fiod**(3fio)

NAME

fio_fiod_status_get – Get status information for a FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_status_get( FIO_APP_HANDLE          app_handle,
                         FIO_DEV_HANDLE          dev_handle,
                         FIO_FIOD_STATUS         *status )
```

DESCRIPTION

This function allows application programs to obtain status information of a registered FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *Status* is a pointer to a **FIO_FIOD_STATUS** structure.

FIO_FIOD_STATUS is defined as:

```
struct
{
        boolean             comm_enabled;/* Is communication enabled? */
        unsigned int        success_rx;  /* Cumulative Successful Rx's */
        unsigned int        error_rx;    /* Cumulative Error Rx's */
                                         /* Frame information array */
        FIO_FRAME_INFO      frame_info[ FIO_TX_FRAME_COUNT ];
} fio_fiod_status;
typedef struct fio_fiod_status FIO_FIOD_STATUS;
```

comm_enabled is a true/false indication of if communications are currently enabled to the FIOD, or not. *success_rx* is a cumulative count of the number of successful responses received by this FIOD. *error_rx* is a cumulative count of the number of error responses (acknowledged errors and timeouts) received by this FIOD. *frame_info* is an array of **FIO_FRAME_INFO** structures, one for each possible request frame. **FIO_TX_FRAME_COUNT** is 128; all possible request frames.

FIO_FRAME_INFO is defined as:

```
struct
{
        FIO_HZ              frequency;      /* Current frame frequency */
        unsigned int        success_rx;     /* Cumulative Successful Rx's */
        unsigned int        error_rx;       /* Cumulative Error Rx's */
        unsigned int        error_last_10;  /* Errors in last 10 frames */
        unsigned int        last_seq;       /* Last frame sequence # */
} fio_frame_info;
typedef struct fio_frame_info FIO_FRAME_INFO;
```

frequency is set to the current request frame frequency being utilized by the FIO API. A value of **FIO_HZ_0** (0) indicates the frame is not currently being sent. *success_rx* is a cumulative count of the number of successful responses received for this request frame. *error_rx* is a cumulative count of the number of error responses (acknowledged errors and timeouts) received for this request frame. *error_last_10* is the count of the number of errors (acknowledged errors and timeouts) for this frame in the transmission of the last 10 frames. *last_seq* is the sequence number associated with the last response received.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

To gain access to the raw response frame data, utilize the **fio_fiod_frame_read**(3fio) interface.

All counters contained in the FIOD status information are four byte unsigned values each with a maximum value of 4,294,967,295. The counters are frozen when they reach the maximum value to prevent rollover. Utilize **fio_fiod_status_reset**(3fio) to reset these counts.

A response frame is only considered successful if it is fully received within the time period defined by the “Handshaking” algorithm in Section 3.3.1.5.3 of the NEMA TS 2 Standard.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_read**(3fio) , **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_frame_schedule_get**(3fio), **fio_fiod_status_reset**(3fio)

NAME

fio_fiod_status_reset – Reset cumulative FIOD status counts

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_status_reset( FIO_APP_HANDLE       app_handle,
                           FIO_DEV_HANDLE       dev_handle )
```

DESCRIPTION

This function provides a method for application programs to reset the communications status counters to 0 (zero) for a registered FIOD.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.

NOTES

All counters returned by **fio_fiod_status_get**(3fio) are reset to 0.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_read**(3fio) , **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_frame_schedule_get**(3fio), **fio_fiod_status_get**(3fio)

NAME

fio_fiod_ts_fault_monitor_get – Get the Fault Monitor State of a **FIOTS1** or **FIOTS2** FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_ts_fault_monitor_get( FIO_APP_HANDLE       app_handle,
                                   FIO_DEV_HANDLE       dev_handle,
                                   FIO_VIEW             view,
                                   FIO_TS_FM_STATE      *fms )
```

DESCRIPTION

This function provides a method to get the current Fault Monitor State of a **FIOTS1** or **FIOTS2** FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOTS1** or **FIOTS2** FIOD. *view* is an indication as to the view of FMS to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *fms* is a pointer to a **FIO_TS_FM_STATE** that will contain the *fms* upon successful completion. The supported values are **FIO_TS_FM_ON** (1), and **FIO_TS_FM_OFF** (0).

The FMS **FIO_VIEW_SYSTEM** *view* is the current *fms* for all application programs that have registered with the **FIOTS1** or **FIOTS2** FIOD. The default FMS is **FIO_TS_FM_ON** (1). If an application program sets the FMS to **FIO_TS_FM_OFF** (0), the application program must subsequently set the FMS back to **FIO_TS_FM_ON** (1) to clear the fault condition.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

All **FIOTS1** or **FIOTS2** registered application programs may set the FMS value. The value used by the FIO API **FIO_TS_FM_OFF** if any application programs set the FMS to off.

The **FIOTS1** or **FIOTS2** must be enabled using **fio_fiod_enable**(3fio) in order for the FMS to become effective.

The FIO API retains ownership of the Fault Monitor output points and does not allow application programs to reserve those output points.

If any application program turns off the Fault Monitor on a FIOD, the FIO API turns off the Fault Monitor output point on that FIOD.

RESTRICTIONS

dev_handle must be a **FIOTS1** or **FIOTS2** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_ts_fault_monitor_set**(3fio) , **fio_fiod_ts1_volt_monitor_set**(3fio), **fio_fiod_ts1_volt_monitor_get**(3fio)

NAME

fio_fiod_ts_fault_monitor_set – Set the Fault Monitor State of a **FIOTS1** or **FIOTS2** FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_ts_fault_monitor_set ( FIO_APP_HANDLE      app_handle,
                                    FIO_DEV_HANDLE      dev_handle,
                                    FIO_TS_FM_STATE     fms )
```

DESCRIPTION

This function provides a method to turn the Fault Monitor output points of **FIOTS1** and **FIOTS2** FIODs on or off.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOTS1** or **FIOTS2** FIOD. *fms* is a **FIO_TS_FM_STATE**. The supported values are **FIO_TS_FM_ON** (1), and **FIO_TS_FM_OFF** (0).

The FMS value sent to the **FIOTS1** or **FIOTS2** is **FIO_TS_FM_OFF** (0), if any application programs turn the Fault Monitor State to off. The default FMS is **FIO_TS_FM_ON** (1). If an application program sets the FMS to **FIO_TS_FM_OFF** (0), the application program must subsequently set the FMS back to **FIO_TS_FM_ON** (1) to clear the fault condition.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

All **FIOTS1** or **FIOTS2** registered application programs may set the FMS value. The value used by the FIO API **FIO_TS_FM_OFF** if any application programs set the FMS to off.

The **FIOTS1** or **FIOTS2** must be enabled using **fio_fiod_enable**(3fio) in order for the FMS to become effective.

The FIO API retains ownership of the Fault Monitor output points and does not allow application programs to reserve those output points.

If any application program turns off the Fault Monitor on a FIOD, the FIO API turns off the Fault Monitor output point on that FIOD.

RESTRICTIONS

dev_handle must be a **FIOTS1** or **FIOTS2** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_ts_fault_monitor_get**(3fio) , **fio_fiod_ts1_volt_monitor_set**(3fio), **fio_fiod_ts1_volt_monitor_get**(3fio)

NAME

fio_fiod_ts1_volt_monitor_get – Get the Volt Monitor State of a **FIOTS1** FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_ts1_volt_monitor_get( FIO_APP_HANDLE       app_handle,
                                   FIO_DEV_HANDLE       dev_handle,
                                   FIO_VIEW             view,
                                   FIO_TS1_VM_STATE     *vms )
```

DESCRIPTION

This function provides a method to get the current Volt Monitor State of a **FIOTS1** FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOTS1** FIOD. *view* is an indication as to the view of VMS to return; **FIO_VIEW_APP** and **FIO_VIEW_SYSTEM** may be specified, to view application program specific data or system data, respectively. *vms* is a pointer to a **FIO_TS1_VM_STATE** that will contain the *vms* upon successful completion. The supported values are **FIO_TS1_VM_ON** (1), and **FIO_TS1_VM_OFF** (0).

The VMS **FIO_VIEW_SYSTEM** *view* is the current *vms* for all application programs that have registered with the **FIOTS1** FIOD. The default VMS is **FIO_TS1_VM_ON** (1). If an application program sets the VMS to **FIO_TS1_VM_OFF** (0), the application program must subsequently set the VMS back to **FIO_TS1_VM_ON** (1) to clear the fault condition.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

All **FIOTS1** registered application programs may set the VMS value. The value used by the FIO API **FIO_TS1_VM_OFF** if any application programs set the VMS to off.

The **FIOTS1** must be enabled using **fio_fiod_enable**(3fio) in order for the VMS to become effective.

The FIO API retains ownership of the Volt Monitor output points and does not allow application programs to reserve those output points.

If any application program turns off the Volt Monitor on a FIOD, the FIO API turns off the Volt Monitor output point on that FIOD.

RESTRICTIONS

dev_handle must be a **FIOTS1** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_ts_fault_monitor_set**(3fio), **fio_fiod_ts_fault_monitor_get**(3fio), **fio_fiod_ts1_volt_monitor_set**(3fio)

NAME

fio_fiod_ts1_volt_monitor_set – Set the Volt Monitor State of a **FIOTS1** FIOD

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_ts1_volt_monitor_set ( FIO_APP_HANDLE      app_handle,
                                    FIO_DEV_HANDLE      dev_handle,
                                    FIO_TS1_VM_STATE    vms )
```

DESCRIPTION

This function provides a method to turn the Volt Monitor output points of a **FIOTS1** FIODs on or off.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *dev_handle* must refer to a **FIOTS1** FIOD. *vms* is a **FIO_TS1_VM_STATE**. The supported values are **FIO_TS1_VM_ON** (1), and **FIO_TS1_VM_OFF** (0).

The VMS value sent to the **FIOTS1** is **FIO_TS1_VM_OFF** (0), if any application programs turn the Volt Monitor State to off. The default VMS is **FIO_TS1_VM_ON** (1). If an application program sets the VMS to **FIO_TS1_VM_OFF** (0), the application program must subsequently set the VMS back to **FIO_TS1_VM_ON** (1) to clear the fault condition.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

All **FIOTS1** registered application programs may set the VMS value. The value used by the FIO API **FIO_TS1_VM_OFF** if any application programs set the VMS to off.

The **FIOTS1** must be enabled using **fio_fiod_enable**(3fio) in order for the VMS to become effective.

The FIO API retains ownership of the Volt Monitor output points and does not allow application programs to reserve those output points.

If any application program turns off the Volt Monitor on a FIOD, the FIO API turns off the Volt Monitor output point on that FIOD.

RESTRICTIONS

dev_handle must be a **FIOTS1** FIOD.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_ts_fault_monitor_set**(3fio), **fio_fiod_ts_fault_monitor_get**(3fio), **fio_fiod_ts1_volt_monitor_get**(3fio)

NAME

fio_fiod_wd_deregister – Deregister for the Watchdog Monitor Service

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_wd_deregister( FIO_APP_HANDLE      app_handle,
                            FIO_DEV_HANDLE      dev_handle )
```

DESCRIPTION

This function allows application programs to deregister for shared control of the Watchdog output point. This application program is taken off the list of application programs required to **fio_fiod_wd_heartbeat**(3fio) in order to toggle the Watchdog output point.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call.

The Watchdog Monitor Service allows a single reserved output point to be toggled; jointly by all application programs registered with that FIOD. This output point is reserved using the **fio_fiod_wd_reservation_set**(3fio) function call. The FIO API restricts the ability to assign the Watchdog output point to the first application program to call this function. The FIO API retains ownership of the Watchdog output point and does not allow application programs to reserve that output point directly. The output point is toggled when all registered application programs call the **fio_fiod_wd_heartbeat**(3fio) function.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
EACCESS	The Watchdog Monitor Service is not currently enabled.

NOTES

The Watchdog output point is established by the first application program calling **fio_fiod_wd_reservation_set**(3fio).

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_wd_register**(3fio), **fio_fiod_wd_reservation_set**(3fio), **fio_fiod_wd_reservation_get**(3fio), **fio_fiod_wd_heartbeat**(3fio)

NAME

fio_fiod_wd_heartbeat – Toggle the output point used for Watchdog Monitoring

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_wd_heartbeat( FIO_APP_HANDLE        app_handle,
                           FIO_DEV_HANDLE        dev_handle )
```

DESCRIPTION

This function provides a method for Watchdog registered application programs to "request" that the FIO API toggle the state of the Watchdog output point. The FIO API only toggles the Watchdog output point if all Watchdog registered application programs have made a call to this function (Watchdog Triggered Condition). Upon a Watchdog Triggered Condition, the FIO API toggles the state of the Watchdog output point within the FIO API. When the FIO API updates the output states of the FIOD, the FIO API clears all previous toggle requests and the Watchdog Triggered Condition so that a new Watchdog Triggered Condition can be generated. The FIO API does not toggle the Watchdog output point more than once per update of the output states on the FIOD.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call.

The Watchdog Monitor Service allows a single reserved output point to be toggled; jointly by all application programs registered with that FIOD. This output point is reserved using the **fio_fiod_wd_reservation_set**(3fio) function call. The FIO API restricts the ability to assign the Watchdog output point to the first application program to call this function. The FIO API retains ownership of the Watchdog output point and does not allow application programs to reserve that output point directly. The output point is toggled when all registered application programs call the **fio_fiod_wd_heartbeat**(3fio) function.

RETURN VALUES

Upon successful completion 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENODEV	The Watchdog output point has not yet been established.
EACCESS	The Watchdog Monitor Service is not currently enabled.

NOTES

RESTRICTIONS

More than one FIOD can have a Watchdog output point toggled via this mechanism. However, each FIOD is only allowed one Watchdog output point.

Only one application program is allowed to establish the Watchdog output point.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_wd_register**(3fio), **fio_fiod_wd_deregister**(3fio), **fio_fiod_wd_reservation_set**(3fio), **fio_fiod_wd_reservation_get**(3fio)

NAME

fio_fiod_wd_register – Register for the Watchdog Monitor Service

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_wd_register( FIO_APP_HANDLE        app_handle,
                          FIO_DEV_HANDLE        dev_handle )
```

DESCRIPTION

This function allows application programs to register for shared control of the Watchdog output point.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call.

The Watchdog Monitor Service allows a single reserved output point to be toggled; jointly by all application programs registered with that FIOD. This output point is reserved using the **fio_fiod_wd_reservation_set**(3fio) function call. The FIO API restricts the ability to assign the Watchdog output point to the first application program to call this function. The FIO API retains ownership of the Watchdog output point and does not allow application programs to reserve that output point directly. The output point is toggled when all registered application programs call the **fio_fiod_wd_heartbeat**(3fio) function.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

The Watchdog output point is established by the first application program calling **fio_fiod_wd_reservation_set**(3fio).

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_wd_deregister**(3fio), **fio_fiod_wd_reservation_set**(3fio), **fio_fiod_wd_reservation_get**(3fio), **fio_fiod_wd_heartbeat**(3fio)

NAME

fio_fiod_wd_reservation_get – Discover the output point used for Watchdog Monitoring

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_wd_reservation_get( FIO_APP_HANDLE  app_handle,
                                 FIO_DEV_HANDLE  dev_handle,
                                 unsigned int    *output_point )
```

DESCRIPTION

This function provides a method which allows application programs to discover the output point used for the Watchdog output of any registered FIOD. The FIO API restricts the ability to assign the Watchdog output point to the first application program to call **fio_fiod_wd_reservation_set**(3fio). All subsequent calls will result in an error. The FIO API retains ownership of the Watchdog output point and does not allow application programs to reserve that output point directly.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *output_point* is a point to an unsigned int that will contain the Watchdog output point number, for this FIOD, upon successful completion. Once the Watchdog output point is established, there is no way to change this assignment. The first application program to assign the Watchdog output point "wins."

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENODEV	The Watchdog output point has not yet been established.
EACCESS	The Watchdog Monitor Service is not currently enabled.

NOTES

None

RESTRICTIONS

Only one application program is allowed to establish the Watchdog output point.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_wd_register**(3fio), **fio_fiod_wd_deregister**(3fio), **fio_fiod_wd_reservation_set**(3fio), **fio_fiod_wd_heartbeat**(3fio)

NAME

fio_fiod_wd_reservation_set – Reserve an output point for Watchdog Monitoring

SYNOPSIS

```
#include        <fio.h>

int fio_fiod_wd_reservation_set( FIO_APP_HANDLE      app_handle,
                                 FIO_DEV_HANDLE      dev_handle,
                                 unsigned int        output_point )
```

DESCRIPTION

This function provides a method which allows application programs to assign the output point used for the Watchdog output of any registered FIOD. The FIO API restricts the ability to assign the Watchdog output point to the first application program to call this function. All subsequent calls will result in an error. The FIO API retains ownership of the Watchdog output point and does not allow application programs to reserve that output point directly.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *dev_handle* is a **FIO_DEV_HANDLE** returned by a previously successful **fio_fiod_register**(3fio) call. *output_point* is a valid output point number for this FIOD. Once the Watchdog output point is established, there is no way to change this assignment. The first application program to assign the Watchdog output point "wins."

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
EEXIST	Watchdog output point has already been established by another application program.
EACCESS	The Watchdog Monitor Service is not currently enabled.

NOTES

None

RESTRICTIONS

Only one application program is allowed to establish the Watchdog output point.

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_outputs_reservation_set**(3fio), **fio_fiod_wd_register**(3fio), **fio_fiod_wd_deregister**(3fio), **fio_fiod_wd_reservation_get**(3fio), **fio_fiod_wd_heartbeat**(3fio)

NAME

fio_hm_deregister – Deregister with the Health Monitor Service

SYNOPSIS

#include <fio.h>

int fio_hm_deregister(FIO_APP_HANDLE *app_handle* **)**

DESCRIPTION

This function is used to deregister this application program with and disable the FIO API Health Monitor Service (each application program has its own unique API Health Monitor timeout).

app_handle is a **FIO_APP_HANDLE** returned from a previously successful call to **fio_register**(3fio).

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
EACCESS	The Health Monitor Service is not currently enabled.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fio_deregister(3fio), **fio_hm_register**(3fio), **fio_hm_heartbeat**(3fio), **fio_hm_fault_reset**(3fio)

NAME

fio_hm_fault_reset – Reset a Health Monitor fault condition

SYNOPSIS

#include <fio.h>

int fio_hm_fault_reset(FIO_APP_HANDLE *app_handle* **)**

DESCRIPTION

This function is used to reset a Health Monitor Service fault condition (timeout). The Health Monitor Service timeout is reset to the last timeout value set via a successful **fio_hm_register**(3fio) call and communications are reestablished with any FIOD that was enabled prior to the fault condition (timeout), as if **fio_fiod_enable**(3fio) was called for each FIOD previously enabled..

app_handle is a **FIO_APP_HANDLE** returned from a previously successful call to **fio_register**(3fio).

If the FIO API Health Monitor timeout expires for an application program, the FIO API disables all FIODs registered by that application program; as if the application program had called **fio_fiod_disable**(3fio) for each FIOD it has registered. The **fio_hm_fault_reset**(3fio) function is utilized to recover from this condition.

An application program is only able to reset its own Health Monitor fault condition and not that of any other application program.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.
EACCESS	The Health Monitor Service is not currently enabled.

NOTES

If an application program resets the API Health Monitor fault condition, then any devices that were disabled due to that condition shall be re-enabled.

If an application program attempts to enable a device using **fio_fiod_enable**(3fio) that has been disabled due to an API Health Monitor fault condition, then the enable operation returns an error and the FIOD remains disabled.

RESTRICTIONS

None

SEE ALSO

fio_deregister(3fio), **fio_hm_register**(3fio), **fio_hm_deregister**(3fio), **fio_hm_heartbeat**(3fio)

NAME

fio_hm_heartbeat – Tell the Health Monitor that the application program is operational

SYNOPSIS

#include <fio.h>

int fio_hm_heartbeat(FIO_APP_HANDLE *app_handle* **)**

DESCRIPTION

This function is used to notify the Health Monitor Service that the application program is operational. The Health Monitor Service timeout is reset to the last timeout value set via a successful **fio_hm_register**(3fio) call.

app_handle is a **FIO_APP_HANDLE** returned from a previously successful call to **fio_register**(3fio).

If the FIO API Health Monitor timeout expires for an application program, the FIO API disables all FIODs registered by that application program, as if the application program had called **fio_fiod_disable**(3fio) for each FIOD it has registered.

The FIO API provides a method, **fio_hm_fault_reset**(3fio), for an application program to reset an FIO API Health Monitor fault condition and allow the FIO API to resume FIOD communications. A call to **fio_hm_heartbeat**(3fio) after a Health Monitor fault has occurred does not reset the Health Monitor fault condition, **fio_hm_fault_reset**(3fio) must be called.

RETURN VALUES

Upon successful completion, 0 is returned. A return value of 1 indicates that a Health Monitor fault condition (timeout exceeded) exists. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
EACCESS	The Health Monitor Service is not currently enabled.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fio_deregister(3fio), **fio_hm_register**(3fio), **fio_hm_deregister**(3fio), **fio_hm_fault_reset**(3fio)

NAME

fio_hm_register – Register with the Health Monitor Service

SYNOPSIS

```
#include        <fio.h>

int fio_hm_register( FIO_APP_HANDLE     app_handle,
                     unsigned int       timeout )
```

DESCRIPTION

This function is used to register this application program with the FIO API Health Monitor Service (each application program has its own unique FIO API Health Monitor *timeout*). This function is also used to modify an application programs Health Monitor *timeout*.

app_handle is a **FIO_APP_HANDLE** returned from a previously successful call to **fio_register**(3fio). *timeout* specifies when the FIO API will declare this application program to be non-operational. *timeout* is in 10ths of a second. *timeout* indicates the maximum allowable time between calls to the FIO API Health Monitor function **fio_hm_heartbeat**(3fio).

Once an application program is registered with the Health Monitor Service, the application program can modify its *timeout* value by calling this function subsequently. This does not reset a fault condition however, **fio_hm_fault_reset**(3fio) must be utilized to reset a Health Monitor fault condition.

If the FIO API Health Monitor *timeout* expires for an application program, the FIO API disables all FIODs registered by that application program, as if **fio_fiod_disable**(3fio) had been called for each one.

The Health Monitor Service may be disabled by calling **fio_hm_deregister**(3fio) or by setting the *timeout* to 0.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.
EACCESS	The Health Monitor Service is not currently enabled.

NOTES

None

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_hm_deregister**(3fio), **fio_hm_heartbeat**(3fio), **fio_hm_fault_reset**(3fio)

NAME

fio_query_fiod – Query to see if a FIOD exists on a port

SYNOPSIS

```
#include        <fio.h>

int fio_query_fiod( FIO_APP_HANDLE          app_handle,
                    FIO_PORT                port,
                    FIO_DEVICE_TYPE         dev )
```

DESCRIPTION

This function is used to see if the indicated FIOD exists on the indicated port.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful call to **fio_register**(3fio). *port* is a **FIO_PORT** that is a valid serial port to which the FIOD may be connected. *dev* is a **FIO_DEVICE_TYPE** that indicates the type of FIOD that is being queried for.

RETURN VALUES

Upon successful completion, 0 is returned to indicate that the requested FIOD does NOT exist on the indicated port. A return value of 1 indicates that the requested FIOD does exist on the indicated port. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.

NOTES

If the FIO API does not have the communications port open at the time of the query, and it is necessary for the FIO API to open the communications port to determine if the FIOD exists, the FIO API closes the communications port after the query is completed.

If the FIO API has the communications port open at the time of the query, and the communications attributes for the FIOD used in the query are not compatible with the current settings on the communications port, the FIO API assumes that the FIOD is not present.

If the FIO API has the communications port open at the time of the query, and FIO API is already successfully completing scheduled communications to the FIOD, the FIO API indicates that the FIOD is present without sending any additional frames to the device.

It is not required for an application program to be registered with a FIOD to perform a query.

RESTRICTIONS

None

SEE ALSO

fio_register(3fio), **fio_fiod_register**(3fio), **fio_fiod_deregister**(3fio)

NAME

fio_query_frame_notify_status – Discover why a response frame notification occurred

SYNOPSIS

```
#include        <fio.h>

int fio_query_frame_notify_status( FIO_APP_HANDLE       app_handle,
                                   FIO_NOTIFY_INFO      *notify_info )
```

DESCRIPTION

This function is used to discover why a notification, via a **FIO_SIGIO** real-time signal, was sent to the application program by the FIO API.

app_handle is a **FIO_APP_HANDLE** returned by a previously successful **fio_register**(3fio) call. *notify_info* is a pointer to a **FIO_NOTIFY_INFO** structure, which will be filled with response frame notification information upon successful completion.

FIO_NOTIFY_INFO is defined as:

```
struct
{
        unsigned int            rx_frame;       /* Response Frame # */
        FIO_FRAME_STATUS        status;         /* Status of response frame */
        unsigned int            seq_number;     /* Sequence Number of frame */
        unsigned int            count;          /* # of bytes in response frame */
        FIO_DEV_HANDLE          fiod;           /* FIOD of response frame */
} fio_notify_info;
typedef struct fio_notify_info FIO_NOTIFY_INFO;
```

rx_frame will be set to the response frame number/type that was received or in error, and is being notified. Valid response frame numbers/types are in the range of 128 – 255. *Status* will be set to an indication as to why the notification occurred. Valid values are: **FIO_FRAME_ERROR** and **FIO_FRAME_RECEIVED**, indicating an error occurred in the reception of the response frame or a valid response frame was received, respectively. *seq_number* is the sequence number given to the response frame that caused the notification. This information may be used to verify a response frame received from **fio_fiod_frame_read**(3fio). *count* is the number of bytes in the raw response frame that was received. *fiod* is the FIOD that sent the response frame causing the notification and may be used in subsequent **fio_fiod**(3fio) calls.

Notification is generated to the application program utilizing the **FIO_SIGIO** (**SIGRTMIN** + 4) real-time signal. The application program using this service must establish a **FIO_SIGIO** handler, prior to calling **fio_fiod_frame_notify_register**(3fio). The default handling for a **FIO_SIGIO** real-time signal is to terminate the process. When the indicated *rx_frame* is received or declared in error, the FIO API will generate a **FIO_SIGIO** real-time signal to the waiting application program. The application program must then perform a **fio_query_frame_notify_status**(3fio) call to discover why a **FIO_SIGIO** real-time signal was generated. The raw data of the most recently received response frame, as indicated by *rx_frame*, may be retrieved using **fio_fiod_frame_read**(3fio), utilizing the values that are returned by this function.

An application program may register notifications for multiple response frames.

RETURN VALUES

Upon successful completion, 0 is returned. A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

EINVAL	One or more arguments are invalid.
ENOMEM	There is not enough memory available for this operation.
EACCESS	The Frame Notification Service is not currently enabled.

NOTES

The **FIO_SIGIO** real-time signal is generated by the FIO API when the indicated *rx_frame* is received for the indicated *fiod*. It is not considered an error to register notification for the same *rx_frame* multiple times. The values of the last call to **fio_fiod_frame_notify_register**(3fio) will be used.

The notification can be set for a one time occurrence or continuous occurrence, based upon the *notify* value passed to **fio_fiod_frame_notify_register**(3fio). If **FIO_NOTIFY_ALWAYS** is passed, the FIO API automatically resets the notification process for the notification of *rx_frame*. If **FIO_NOTIFY_ONCE** is passed, the FIO API automatically disables future notification (as if **fio_fiod_frame_notify_deregister**(3fio) has been called) of the *rx_frame*.

Due to timing and possible race conditions, it is possible to receive a *seq_number* via this function that does not match with the *seq_number* returned by **fio_fiod_frame_read**(3fio). This should not be considered an error. *seq_number* returned by this function should be used for informational purposes only.

Real-time **FIO_SIGIO** signals are queued to guarantee that all desired notifications are received. This function returns the queued notifications in a FIFO order.

The response frame sequence number is a four byte unsigned value that rolls over after the maximum value.

RESTRICTIONS

None

SEE ALSO

fio_fiod_register(3fio), **fio_fiod_enable**(3fio), **fio_fiod_disable**(3fio), **fio_fiod_frame_schedule_set**(3fio), **fio_fiod_frame_schedule_get**(3fio), **fio_fiod_frame_size**(3fio), **fio_fiod_frame_read**(3fio), **fio_fiod_frame_notify_register**(3fio), **fio_fiod_frame_notify_deregister**(3fio)

NAME

fio_register – Register with the FIO API

SYNOPSIS

#include <fio.h>

FIO_APP_HANDLE fio_register( )

DESCRIPTION

The **fio_register**(3fio) function is used to register this application program with the FIO API. A process must register with the FIO API before any of the FIO API services may be utilized.

RETURN VALUES

Upon successful completion a valid **FIO_APP_HANDLE** is returned. This handle must be used in all subsequent **fio**(3fio) function calls.

A negative return value indicates an error has occurred. In this case, *errno* is set with the error that has occurred.

ERRORS

Errors are indicated by negative return values. Error codes returned in *errno*:

ENOMEM There is not enough memory available for this operation.

NOTES

An application program is defined as a single Linux Process, as defined by a unique process id. If output points or channels need to be shared between processes, this sharing must be done at a higher level. Writing of an output point or channel is exclusive to a Linux Process. If two or more Linux Processes want to write to the same output point or channel; this sharing must be solved by the user of the FIO API. The FIO API does not directly support output point and channel sharing.

The FIO API assumes it has exclusive access to the serial communications ports of the ATC Engine Board that are designated for FIODs.

The FIO API supports all cabinet architectures and associated FIOD types as listed in the ATC Controller Standard Section 8.

The process of application program registration does not cause the FIO API to perform any communications with a FIOD.

RESTRICTIONS

None

SEE ALSO

fio_deregister(3fio), **fio_fiod_register**(3fio), **fio_hm_register**(3fio)

4.3 Utility Functions

This section specifies functions that are part of the API but are not considered a part of the API managers.

4.3.1 Time of Day Functions

This Time of Day (TOD) library provides a common interface for an ATC controller's time of day information. The TOD functionality includes keeping track of time from one of several sources, storing time zone information, handling daylight saving transactions and providing signals to the calling process whenever the clock ticks or the local time is changed. Operationally, they can be grouped as shown below.

- Set/Get Time Functions – tod_set and tod_get.
- Daylight Saving Time (DST) Functions – tod_set_dst_state, tod_get_dst_state, tod_get_dst_info, and tod_set_dst_info.
- Time Source and Signaling Functions – tod_get_timesrc, tod_set_timesrc, tod_get_timesrc_freq, tod_request_tick_signal, tod_cancel_tick_signal, tod_request_onchange_signal, and tod_cancel_onchange_signal.

The *timeval* structure defined in the Linux header file "sys/time.h" is used by the API functions tod_set and tod_get in order to set and get the current time (see functions tod_set and tod_get). This structure is defined as:

```
struct timeval {        /* defined in sys/time.h */
        long tv_sec;    /* seconds */
        long tv_usec;   /* microseconds */
};
```

Software developers should take special care if they intend to set the system time from an application program. They must consider that other application programs on the controller unit may also be setting the time. If precautions are not taken, a jump in time forward or backward can cause time triggered events in application programs to be missed or repeated, or to occur at a different interval than originally intended.

The API provides functions to support the handling of Daylight Saving Time. A variety of methods in which to specify parameters for DST around the world are provided (see http://webexhibits.org/daylightsaving/g.html and other time Web sites to see world DST periods). Each method defines the date and time in which DST begins and ends, as well as the number of seconds to adjust (+/-). The methods are as follows:

Absolute Method. The time to make the transition is specified by the number of seconds from midnight UTC of January 1, 1970 (called “epoch”).

Generic Method #1. The time to make the transition is specified by the month, day of month and the time.

Generic Method #2. The time to make the transition is specified by the month, day of month and the time. In this case, the day of the month is specified as the occurrence of the day of the week starting on-or-after a specified day of the month.

Generic Method #3. The time to make the transition is specified by the month, day of month and the time. In this case, the day of the month is specified as the occurrence of a day of week starting on-or-before a specified day of the month and moving backwards.

The daylight saving time information is contained in the structure *dst_info.* A typedef, *dst_info_t*, is provided for convenience.

```
typedef struct dst_info {
	char type;  // 0=absolute, 1=generic
	union dst_types_union {
		struct dst_absolute_struct {
			int secs_from_epoch_to_transition;
			int seconds_to_adjust;
		} absolute;
		struct dst_generic_struct {
			char month;
			char dom_type;  // 0=dom, 1=forward_occurances, 2=reverse_occurances
			union dst_gen_dom_union {
				char dom;
				// ex: second Saturday of month
				// ex: first Sunday on or after oct. 9
				struct dst_gen1_struct {
					char dow;  // day of week (sun-sat)
					char occur;  // number of occurrences
					char on_after_dom;	// day of month
				} forward_occurances_of_dow;
				// ex: second to last Thursday of month
				// ex: first Sunday on or before oct. 9
				struct dst_gen2_struct {
					char dow;  // day of week (sun-sat)
					char occur;  // number of occurrences
					char on_before_dom	// day of month
				} reverse_occurances_of_dow;
			} generic_dom;
			int secs_from_midnight_to_transition;
			int seconds_to_adjust;
		} generic;
	} begin, end;
} dst_info_t;
```

The *dst_info* structure contains two identical unions named *begin* and *end*. The *begin* union contains the information necessary to determine when daylight saving should begin by adjusting the time, and the *end* union contains the information necessary to determine when daylight saving should end by re-adjusting the time. The unions contain two structures named *absolute* and *generic*. The *type* member of the struct *dst_info* indicates which format is used for *begin* and *end*. The *type* member shall be 0 for *absolute* or 1 for *generic*. The *absolute* structure contains the exact date and time at which the beginning/ending adjustment should be made and by how many seconds the time should be adjusted. The *generic* structure defines the DST begin and end points in a format that is valid yearly. It contains the month in which the beginning/ending adjustment should be made and a union named *generic_dom* (short for generic day of month) that contains the information to determine the day of the month on which the beginning/ending adjustment will take place. The information in the generic day of month union determines a particular day of the month by using *dom* to indicate a specific day of the month, or by using *forward_occurances_of_dow* or *reverse_occurances_of_dow* to indicate a day of the week that occurs a number of times before or after a specific day of the month. The *dom_type* member of the union dst_gen_dom_union shall be a 0, 1, or 2 determining whether the *dom* member, *forward_occurances_of_dow* member or *reverse_occurances_of_dow* union member respectively is used. The *dom*, *on_after_dom*, and *on_before_dom* members specify a day of the month from 1 to 31 inclusive. The *occur* member shall be 1 or greater, indicating the number of times the particular day of the week, identified by the *dow* member (0 – 6, 0 being Sunday), shall occur to determine the day of the month when the daylight saving adjustment shall take place. The *dst_info* structure is used in the functions tod_set_dst_state, tod_get_dst_state, tod_set_dst_info and tod_get_dst_info.

ATC controller units may support multiple time sources (or clock references) to meet the needs of different agencies or different equipment configurations. The time source is the method by which the operating system tracks real time. The enumerated data type *TOD_TIMESRC_ENUM* defines time sources which are commonly available. Note that some of these time sources are optional per the ATC Controller Standard.

```
enum {
        TOD_TIMESRC_LINESYNC,
        TOD_TIMESRC_RTCSQWR,
        TOD_TIMESRC_CRYSTAL,
        TOD_TIMESRC_EXTERNAL1,
        TOD_TIMESRC_EXTERNAL2
} TOD_TIMESRC_ENUM;
```

The enumerations of *TOD_TIMESRC_ENUM* are defined as follows:

TOD_TIMESRC_LINESYNC indicates that clock ticks are determined by the linesync signal from an ATC power supply;

TOD_TIMESRC_RTCSQWR indicates that clock ticks are determined by a square wave output of an internal hardware RTC;

TOD_TIMESRC_CRYSTAL indicates that clock ticks are determined by the processor clock crystal;
TOD_TIMESRC_EXTERNAL1 indicates that clock ticks are derived from or synchronized to some external method which is implementation dependent (e.g. GPS, NTP, etc.); and
TOD_TIMESRC_EXTERNAL2 is a second source also derived from or synchronized to some external method which is implementation dependent (e.g. GPS, NTP, etc.).

Often application programs that are time oriented utilize scheduling mechanisms based on operating system “clock ticks.” Since the frequency of a clock tick will vary depending on the time source (e.g. 60 Hz based on linesync or many megahertz based on the processor clock crystal), software developers must consider this in the design of their application programs. It should also be noted that a fraction of a clock tick may be gained or lost when the time source is changed.

The API Standard provides the following functions so that application programs may manage the time source: tod_get_timesrc, tod_set_timesrc, tod_get_timesrc_freq, tod_request_tick_signal, tod_cancel_tick_signal, tod_request_onchange_signal and tod_cancel_onchange_signal.

The header file, “tod.h,” defines all of the TOD function calls in a library named “libtod.so.” The header file also includes sys/time.h.

NAME

tod_cancel_onchange_signal – cancels local time changed signals

SYNOPSIS

```
#include        <tod.h>

int tod_cancel_onchange_signal( )
```

DESCRIPTION

The **tod_cancel_onchange_signal**(3tod) library call cancels any local time change signal from being sent to the calling process.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

Any errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

If the process that has requested this signal dies, the equivalent to this call will be performed automatically to release any necessary resources.

RESTRICTIONS

None

SEE ALSO

tod_request_onchange_signal(3tod)

NAME

tod_cancel_tick_signal – Cancel signal request for TOD ticks

SYNOPSIS

```
#include        <tod.h>

int tod_cancel_tick_signal( )
```

DESCRIPTION

The **tod_cancel_tick_signal**(3tod) library call cancels any time of day clock tick signal from being sent to the calling process.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

Any errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

If the process that has requested this signal dies, the equivalent to this call will be performed automatically to release any necessary resources.

RESTRICTIONS

None

SEE ALSO

tod_request_tick_signal(3tod)

NAME

tod_get – Get the current time, time zone offset, and dst offset

SYNOPSIS

#include <tod.h>

int tod_get(struct timeval *_tv_, int *_tzsec_offset_, int *_dst_offset_)

DESCRIPTION

The **tod_get**(3tod) library call retrieves the current date, time, time zone and the offset being applied due to daylight saving time.

Parameters:

tv receives the local time as a number of seconds since midnight of January 1, 1970.

tzsec_offset is the time zone offset in seconds from the UTC time to the local standard time in the range of -43,200 to +43,200. The *dst_offset* is the offset in seconds from the local standard time to the local daylight saving time corresponding to the affect daylight saving time is currently having on the local time.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

Any errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes. For all practical purposes the user application should only be concerned with success or fail.

NOTES

If daylight saving time is enabled, this time will reflect the current daylight saving time. The UTC/GMT time in seconds since midnight, January 1, 1970 can be determined by subtracting the *tzsec_offset* from the tv_sec member in *tv* and then further subtracting the *dst_offset* value and storing the result back in the tv_sec member.

If any of the parameters are passed a null pointer, those parameter will not have their values set, but the other parameters will be still be set appropriately and an error shall not be returned.

RESTRICTIONS

None

SEE ALSO

tod_set(3tod)

NAME

tod_get_dst_info – Get daylight saving time information

SYNOPSIS

#include <tod.h>

int tod_get_dst_info(const dst_info_t *_dst_info_)

DESCRIPTION

The **tod_get_dst_info**(3tod) library call gets the information that determines when DST adjustments will occur when DST is enabled. The dst_info parameter must point to a structure where the current daylight saving information will be placed.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

EINVAL *dst_info* is a null pointer.

Any other errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

None

RESTRICTIONS

None

SEE ALSO

tod_set_dst_info(3tod)

NAME

tod_get_dst_state – Return whether DST is enabled or disabled

SYNOPSIS

#include <tod.h>

int tod_get_dst_state( )

DESCRIPTION

The **tod_get_dst_state**(3tod) library call returns zero if daylight saving time is disabled and one if daylight saving time is enabled.

RETURN VALUES

On success, 0 or 1 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

Any errors shall be due to a system call error in the library implementation, in which case, the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

This function returns whether DST is enabled, but to determine if DST is currently in effect and having an impact on the local time, **tod_get**(3tod) can be used to obtain the dst_offset, which will be non-zero if DST is in effect.

RESTRICTIONS

None

SEE ALSO

tod_set_dst_state(3tod)**, tod_set_dst_info**(3tod)**, tod_get_dst_info**(3tod)

NAME

tod_get_timesrc – Get the time source that affects the time

SYNOPSIS

```
#include        <tod.h>

int tod_get_timesrc( )
```

DESCRIPTION

The **tod_get_timesrc**(3tod) library call returns the current time source as an integer value corresponding to a value in the **TOD_TIMESRC_ENUM** enumeration, defined as follows:

```
enum {
        TOD_TIMESRC_LINESYNC,
        TOD_TIMESRC_RTCSQWR,
        TOD_TIMESRC_CRYSTAL,
        TOD_TIMESRC_EXTERNAL1,
        TOD_TIMESRC_EXTERNAL2
} TOD_TIMESRC_ENUM;
```

The **TOD_TIMESRC_ENUM** values are defined as follows:

- **TOD_TIMESRC_LINESYNC** - time ticks are determined by linesync signal from ATC power supply
- **TOD_TIMESRC_RTCSQWR** - time ticks are determined by square wave output of an RTC
- **TOD_TIMESRC_CRYSTAL** - time ticks are determined by the processor crystal
- **TOD_TIMESRC_EXTERNAL1** - time is derived/synchronized by some external method, which is implementation dependent (e.g. GPS, etc)
- **TOD_TIMESRC_EXTERNAL2** - time is derived/synchronized by some external method, which is implementation dependent (e.g. NTP, etc)

RETURN VALUES

tod_get_timesrc(3tod) returns a positive value is returned corresponding to an entry in the **TOD_TIMESRC_ENUM** enumeration or -1 if an error occurred with *errno* set appropriately.

ERRORS

Any errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

None

RESTRICTIONS

None

SEE ALSO

tod_set_timesrc(3tod)

NAME

tod_get_timesrc_freq – Return the input frequency of the time source

SYNOPSIS

#include <tod.h>

int tod_get_timesrc_freq( )

DESCRIPTION

The **tod_get_timesrc_freq**(3tod) library call returns the frequency in Hz of the time source tick. This function returns the expected (ideal) frequency of the current time source, which should be very close to the actual tick frequency, though the actual frequency may differ by whatever error the current time source may experience (i.e. due to an inaccurate crystal, drift during the day in the line frequency, etc.).

RETURN VALUES

On success, the frequency in Hz is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

Any errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes

NOTES

None

RESTRICTIONS

None

SEE ALSO

tod_request_tick_signal(3tod)

NAME

tod_request_onchange_signal – Request local time changed signals

SYNOPSIS

#include <tod.h>

int tod_request_onchange_signal(int *signalnum***)**

DESCRIPTION

The **tod_request_onchange_signal**(3tod) library call requests that the signal *signalnum* be sent to the calling process whenever the local time is changed by any source other than the time tick source. This includes a signal being sent when local time changes due to a daylight saving time adjustment.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

EINVAL *signalnum* is not a valid signal.

Any other errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes

NOTES

If a signal has already been requested by the calling process when this library function is called, the previous signal being sent will be replaced with the signal specified by *signalnum*.

RESTRICTIONS

None

SEE ALSO

tod_cancel_onchange_signal(3tod)

NAME

tod_request_tick_signal – Request a signal on each TOD tick

SYNOPSIS

#include <tod.h>

int tod_request_tick_signal(int *signalnum***)**

DESCRIPTION

The **tod_request_tick_signal**(3tod) library call requests that the signal *signalnum* be sent to the calling process at each tick of the time of day clock. The frequency of the time of day clock can be determined by calling **tod_get_timesrc_freq**(3tod).

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

EINVAL *signalnum* is not a valid signal.

Any other errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes

NOTES

If a signal has already been requested by the calling process when this library function is called, the previous signal being sent will be replaced with the signal specified by *signalnum*.

RESTRICTIONS

None

SEE ALSO

tod_cancel_tick_signal(3tod)**, tod_get_timesrc_freq**(3tod)

NAME

tod_set – Set the system date, time, and time zone (local time)

SYNOPSIS

#include <tod.h>

int tod_set(const struct timeval *_tv_, const int *_tzsec_offset_)

DESCRIPTION

The **tod_set**(3tod) library call sets the system date, time and time zone.

The *tv* parameter is a pointer to the timeval structure that contains the seconds and microseconds since Midnight of January 1, 1970 until the desired current local time. The *tzsec_offset* parameter is a pointer to a value representing the time zone offset in seconds from the UTC time to the local standard time in the range of -43,200 to +43,200. The *tzsec_offset* parameter has been designed to be compatible with NTCIP 1201 v2.32.

Null can be passed for either parameter so that either only the time is set or only the time zone is set. If the *tzsec_offset* parameter is null and the *tv* parameter is valid, the time will be changed, but the time zone offset will remain the same. If the *tv* parameter is null and the *tzsec_offset* parameter is valid, the time zone will be changed and the time will also be changed according to the change in time zone.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

EINVAL *tv* and *tzsec_offset* are null.
EINVAL *tzsec_offset* outside of the acceptable range.
Any other errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

When this function is called with daylight saving time enabled, if the time is changed from a time where daylight saving was not in effect to a time where daylight saving is in effect, then the dst_offset that is retrieved by **tod_get**(3tod) will be adjusted according to the seconds_to_adjust value in the struct dst_info set by **tod_set_dst_info**(3tod) library call. Likewise, if while DST is enabled, the time is changed so that DST goes from being in effect to no longer being in effect, dst_offset will be adjusted to zero.

RESTRICTIONS

None

SEE ALSO

tod_get(3tod)

NAME

tod_set_dst_info – Set the daylight saving time information

SYNOPSIS

#include <tod.h>

int tod_set_dst_info(dst_info_t *_dst_info_)

DESCRIPTION

The **tod_set_dst_info**(3tod) library call sets the information determining the range of time when daylight saving time will be in effect, and the amount the time will be adjusted when entering and exiting that range. The *dst_info* parameter is a pointer to the dst_info_t defined several pages above.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

EINVAL a value of *dst_info* does not correspond to the allowable values defined for the structure members.

Any other errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

None

RESTRICTIONS

None

SEE ALSO

tod_get_dst_state(3tod)**, tod_set_dst_state**(3tod)**, tod_get_dst_info**(3tod)

NAME

tod_set_dst_state – Enable or disable daylight saving time

SYNOPSIS

#include <tod.h>

int tod_set_dst_state(int *enabled***)**

DESCRIPTION

The **tod_set_dst_state**(3tod) library call enables daylight saving time if *enabled* is true; otherwise daylight saving time is disabled.

When daylight saving time is enabled, daylight saving time is considered to be in effect when the local date and time are within the beginning and ending dates and times determined by the dst_info. When daylight saving time is enabled and in effect, the local time is adjusted by the dst_offset, which is determined from the seconds_to_adjust value in the dst_info structure at the time when dst_info first becomes in effect. When the current date and time passes the ending date and time determined by the dst_info, the local time is adjusted by the seconds_to_adjust value and the dst_offset is set to 0.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

Any errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

None

RESTRICTIONS

None

SEE ALSO

tod_get_dst_state(3tod)**, tod_set_dst_info**(3tod)**, tod_get_dst_info**(3tod)

NAME

tod_set_timesrc – Set the time source that affects the time

SYNOPSIS

#include <tod.h>

int tod_set_timesrc(int *timesrc***)**

DESCRIPTION

The **tod_set_timesrc**(3tod) library call sets the current time source to the **TOD_TIMESOURCE_ENUM** enum value specified by the *timesrc* parameter by switching which time source is used. See the **tod_get_timesrc**(3tod) for the definition and description of the **TOD_TIMESOURCE_ENUM**.

RETURN VALUES

On success, 0 is returned. On error, -1 is returned with *errno* is set appropriately.

ERRORS

EINVAL hardware does not support the time source requested.

Any other errors shall be due to a system call error in the library implementation, in which case the values of *errno* shall correspond to the standard Linux system call error codes.

NOTES

None

RESTRICTIONS

None

SEE ALSO

tod_get_timesrc(3tod)

APPENDICES

Appendix A: Traceability Matrix

The following table shows the relationship between the user needs, the requirements and the functions for the ATC Application Programming Interface. User needs and requirements have been given identifiers using the form ***dip[n]***. Where: ***d*** is the initials of the document where the need/requirement is stated; ***i*** is either “N” for user need or “R” for requirement; ***p*** refers to the paragraph where the need/requirement is found; and ***n*** is the number of the item within the paragraph. An identifier “APIN2.1[3]” would refer to the third user need stated in section 2.1 of the API Standard. The associated functions listed are documented in Section 4 of the standard.

User Need ID	User Need Description	Requirement ID	Requirement Description	Verification / API Function
APIN2.1[1]	When combined with the ATC operating system, the API forms an open architecture software (SW) platform that acts as a universal interface between application programs and the ATC controller units.	APIR3.4[1]	The API shall operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard.	This is an operational requirement for the API software.
APIN2.1[2]	The API, when used with the ATC O/S, facilitates portability by requiring only modest efforts on the part of the developer such as recompiling and linking source code for a particular processor.	APIR3.4[2]	The API function calls shall be specified using the C programming language as described by "ISO/IEC 9899:1999" commonly referred to as the C99 Standard.	Section 4 describes the API accordingly.
		APIR3.5.2[2]	If API functions have a similar operation to existing Linux functions, they shall have a similar name and argument style to those functions to the extent possible without causing compilation issues.	See fpui_close(3fpui) & close(2), fpui_open(3fpui) & open(2), fpui_read(3fpui) & read(2), fpui_write(3fpui) & write(2)
		APIR3.5.2[3]	The API function names shall be lower case.	Section 4 defines the API functions in lower case.
		APIR3.5.2[4]	API functions shall use the Linux "errno" error notification mechanism if an error indication is expected for a function.	Section 4 API functions are defined accordingly.
		APIR3.5.2[5]	The API shall be loadable as an ELF (Executable and Linking Format) library	This is an operational requirement for the API software.
APIN2.1[3]	As shown, the API must provide management functions for the Front Panel and the Field I/O Devices.	APIR3.1.1[1]	The API shall provide a text-based user interface capability to allow application programs running concurrently on an ATC controller unit to share the controller's Front Panel display.	fpui_open(3fpui)
		APIR3.1.2[1]	The API shall assume it has exclusive access to the serial communications ports of the ATC Engine Board that are designated for Field I/O Devices.	This is an operational requirement for the API software. The API functions in Section 4.2 are defined accordingly.
APIN2.1[4]	As shown in Figure 5, both users and application programs use the API to interface to ATC controller units.	APIR3.1.1[1]	The API shall provide a text-based user interface capability to allow application programs running concurrently on an ATC controller unit to share the controller's Front Panel display.	fpui_open(3fpui)
		APIR3.1.2[1]	The API shall assume it has exclusive access to the serial communications ports of the ATC Engine Board that are designated for Field I/O Devices.	This is an operational requirement for the API software. The API functions in Section 4.2 are defined accordingly.
APIN2.1[5]	Functions in the API Software Layer access the ATC unit through the functions in the ATC BSP Layer.	APIR3.4[1]	The API shall operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard.	This is an operational requirement for the API software.

User Need ID	User Need Description	Requirement ID	Requirement Description	Verification / API Function
APIN2.1.2[1]	This requires the API to support a user interface that provides Operational Users with the ability to select from a set of active application programs on the ATC controller unit and to interact with the programs individually.	APIR3.1.1.1[1]	The API shall provide a window selection screen called the Front Panel Manager Window from which Operational Users may select a window to have focus.	fpui_open(3fpui)
		APIR3.1.1.1[2]	Application names associated with each window shall be listed.	fpui_open(3fpui)
		APIR3.1.1.1[3]	The application names shall be limited to 16 characters.	fpui_open(3fpui)
		APIR3.1.1.1[4]	If there is no application program associated with a window, the window number shall be listed with a blank application name.	fpui_open(3fpui)
		APIR3.1.1.1[5]	The default Front Panel Manager Window size shall be 8 lines x 40 characters with the format as shown in Figure 7.	fpui_open(3fpui)
		APIR3.1.1.1[6]	If the Operational User has not set the default window, the Front Panel Manager Window shall be the default window.	This is an operational requirement for the API software.
		APIR3.1.1.1[7]	The default window shall be settable by the Operational User from the Front Panel Manager Window by pressing {*,[0-F],<ENT>}.	This is an operational requirement for the API software.
		APIR3.1.1.1[8]	The Operational User shall be capable of setting the default window to the Front Panel Manager Window by pressing {*,<ENT>} from the Front Panel Manager Window.	This is an operational requirement for the API software.
		APIR3.1.1.1[9]	The default window shall be designated by a star “*” character next to the window number.	This is an operational requirement for the API software.
		APIR3.1.1.1[10]	The Operational User shall be able to put the Front Panel Manager Window in focus by pressing {**,<ESC>} from the keypad on the controller’s Front Panel regardless of the application program in operation.	This is an operational requirement for the API software.
		APIR3.1.1.1[11]	The Operational User shall be able to enter {**} by pressing an asterisk (*) twice within a 1.0 second time period.	This is an operational requirement for the API software.
		APIR3.1.1.1[12]	If the {**} sequence is not completed within the 1.0 second time period or if the {**} sequence is not followed by <ESC> character within a 1.0 second time period, then the characters shall be interpreted as individual “*” characters.	This is an operational requirement for the API software.
		APIR3.1.1.1[13]	The Operational User shall have the capability to put a window in focus that is assigned to an application program by pressing {[0-F]} from the Front Panel Manager Window.	This is an operational requirement for the API software.

User Need ID	User Need Description	Requirement ID	Requirement Description	Verification / API Function
		APIR3.1.1.1[14]	The only possible window selections for focus from the Front Panel Manager Window shall be itself, the ATC Configuration Window (see Section 3.2), or a window assigned to an application program.	This is an operational requirement for the API software.
		APIR3.1.1.1[15]	If the Front Panel Manager Window is the default window, no asterisk shall be displayed next to any application name in the Front Panel Manager Window	This is an operational requirement for the API software.
		APIR3.1.1.1[16]	The Operational User shall be able to put the ATC Configuration Window in focus by pressing {<NEXT>} in the Front Panel Manager Window.	This is an operational requirement for the API software.
		APIR3.1.1.1[17]	The top two lines and bottom line of the Front Panel Manager Window shall be fixed as shown in Figure 7.	This is an operational requirement for the API software.
		APIR3.1.1.1[18]	The number of lines between the second line and bottom lines used for displaying window names shall vary according to the size of the ATC display.	This is an operational requirement for the API software.
		APIR3.1.1.1[19]	The Operational User shall be able to scroll up and down the names of the windows in the Front Panel Manager Window one line at a time using the up and down arrow keys of the controller keypad.	This is an operational requirement for the API software.
APIN2.1.2[2]	The ATC Controller Standard describes the physical characteristics and the character set to be supported.	APIR3.4[1]	The API shall operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard.	This is an operational requirement for the API software.
		APIR3.1.1.2[28]	Application programs shall be able to interpret all ATC controller keys as individual key codes.	fpui_open(3fpui), fpui_set_keymap(3fpui), fpui_get_keymap(3fpui), fpui_del_keymap(3fpui), fpui_keymap(3fpui)
		APIR3.1.1.2[29]	The escape sequences representing keys that do not have standard ASCII character codes on an ATC controller shall be mapped to specific character codes in the API as shown in Table 1.	fpui_set_keymap(3fpui), fpui_get_keymap(3fpui), fpui_del_keymap(3fpui), fpui_keymap(3fpui)
APIN2.1.2[3]	The API must also provide C functions that allow application programs to access and control the Front Panel Interface programmatically.	APIR3.1.1.2[1]	The API shall provide a function to return the dimensions of a window in terms of number of lines and number of columns.	fpui_get_window_size(3fpui)
		APIR3.1.1.2[2]	The API shall provide a function to open a window and register a name for display on the Front Panel Manager Window.	fpui_open(3fpui)

		APIR3.1.1.2[3]	An application program shall be able to open multiple windows providing the windows resources are available.	fpui_open(3fpui)
		APIR3.1.1.2[4]	The API shall provide the ability for an application program to reserve exclusive access to the Aux Switch (see ATC Controller Standard, Section 7.1.4).	fpui_open_aux_switch(3fpui), fpui_close_aux_switch(3fpui)
		APIR3.1.1.2[5]	An application program that has reserved exclusive access to the AUX Switch shall maintain exclusive access to the switch even if the application program has no window in focus.	fpui_open_aux_switch(3fpui), fpui_close_aux_switch(3fpui)
		APIR3.1.1.2[6]	The API shall provide a function to close a window and release the resource for other application programs.	fpui_close(3fpui)
		APIR3.1.1.2[7]	The API shall provide a function or set of functions to set the attributes of a Front Panel display as described in the ATC Controller Standard, Section 7.1.4.	

fpui_set_character_blink(3fpui),
fpui_set_backlight(3fpui),
fpui_set_cursor_blink(3fpui),
fpui_set_reverse_video(3fpui),
fpui_set_underline(3fpui),
fpui_set_auto_wrap(3fpui),
fpui_set_auto_repeat(3fpui),
fpui_set_cursor(3fpui),
fpui_set_auto_scroll(3fpui),
fpui_reset_all(3fpui)

		APIR3.1.1.2[8]	The API shall provide a function or set of functions to return the attributes of a Front Panel display as described in the ATC Controller Standard, Section 7.1.4.	

fpui_get_attributes(3fpui),
fpui_get_character_blink(3fpui),
fpui_get_backlight(3fpui),
fpui_get_cursor_blink(3fpui),
fpui_get_reverse_video(3fpui),
fpui_get_underline(3fpui),
fpui_get_auto_wrap(3fpui),
fpui_get_auto_repeat(3fpui),
fpui_get_cursor(3fpui),
fpui_get_auto_scroll(3fpui)

		APIR3.1.1.2[9]	The API shall provide a function that is used to determine if there is data in the input buffer of a window.	fpui_poll(3fpui)
		APIR3.1.1.2[10]	The API shall provide a function to read a queued character or key code from the input buffer of a window.	fpui_read_char(3fpui)
		APIR3.1.1.2[11]	The API shall provide a function to write a character to the current cursor position of a window.	fpui_write_char(3fpui)
		APIR3.1.1.2[12]	The API shall provide a function to write a character to a window at a position defined by column and line number.	fpui_write_char_at(3fpui)
		APIR3.1.1.2[13]	The API shall provide a function to write a string to a window at the current cursor position.	fpui_write_string(3fpui)
		APIR3.1.1.2[14]	The API shall provide a function to write a string to a window at a starting position defined by column number and line number.	fpui_write_string_at(3fpui)
		APIR3.1.1.2[15]	The API shall provide a function to write a buffer of characters to a window at the current cursor position.	fpui_write(3fpui)
		APIR3.1.1.2[16]	The API shall provide a function to write a buffer of characters to a window at a starting position defined by column number and line number.	fpui_write_at(3fpui)
		APIR3.1.1.2[17]	The API shall provide a function to set the cursor position of a window defined by column and line number.	fpui_set_cursor_pos(3fpui)
		APIR3.1.1.2[18]	The API shall provide a function to return the cursor position of the window defined by column and line number.	fpui_get_cursor_pos(3fpui)
		APIR3.1.1.2[19]	If a window was registered with access to the Aux Switch, the API shall provide a function to return its status.	fpui_read_aux_switch(3fpui)
		APIR3.1.1.2[20]	The API shall provide a function to compose special characters as described by the ATC Controller Standard, Section 7.1.4.	fpui_define_special_char(3fpui)
		APIR3.1.1.2[21]	The API shall support the display of a composed character in the same manner as any other valid character.	fpui_display_special_char(3fpui)
		APIR3.1.1.2[22]	The API shall provide a function to clear a window that operates on a window whether it is in or out of focus.	fpui_clear(3fpui)
		APIR3.1.1.2[23]	The API shall provide a function to refresh a window that operates on a window whether it is in or out of focus.	fpui_refresh(3fpui)

		APIR3.1.1.2[24]	The bell of the controller's Front Panel shall be activated only if a bell character, ^G (hex value 07), is sent from an application program which has a window that has focus.	This is an operational requirement for the API software.
		APIR3.1.1.2[25]	If a bell character is sent from an application program that does not have a window that has focus, the bell character shall be ignored by the API.	This is an operational requirement for the API software.
		APIR3.1.1.2[26]	The API shall a illuminate or extin controller's displa through a window	
fpui_set_backlight(3fpui), fpui_get_backlight(3fpui), fpui_set_backlight_timeout(3fpui)				
		APIR3.1.1.2[27]	Display configuration and inquiry command codes (escape sequences) specified in the ATC Controller Standard, Section 7.1.4, shall be supported as separate functions in the API.	
fpui_get_attributes(3fpui), fpui_set_character_blink(3fpui), fpui_get_character_blink(3fpui), fpui_set_backlight(3fpui), fpui_get_backlight(3fpui), fpui_set_cursor_blink(3fpui), fpui_get_cursor_blink(3fpui), fpui_set_reverse_video(3fpui), fpui_get_reverse_video(3fpui), fpui_set_underline(3fpui), fpui_get_underline(3fpui), fpui_set_auto_wrap(3fpui), fpui_get_auto_wrap(3fpui), fpui_set_auto_repeat(3fpui), fpui_get_auto_repeat(3fpui), fpui_set_cursor(3fpui), fpui_get_cursor(3fpui), fpui_set_auto_scroll(3fpui), fpui_get_auto_scroll(3fpui), fpui_reset_all(3fpui)				
		APIR3.1.1.2[28]	Application programs shall be able to interpret all ATC controller keys as individual key codes.	fpui_set_keymap(3fpui), fpui_get_keymap(3fpui), fpui_del_keymap(3fpui), fpui_keymap(3fpui)

		APIR3.1.1.2[29]	The escape sequences representing keys that do not have standard ASCII character codes on an ATC controller shall be mapped to specific character codes in the API as shown in Table 1.	fpui_set_keymap(3fpui), fpui_get_keymap(3fpui), fpui_del_keymap(3fpui), fpui_keymap(3fpui)
		APIR3.1.1.2[30]	The ATC Controller Standard, Section 7.1.4, describes a graphics interface to the Front Panel's display. The API shall support the operation of the graphics commands on a window only if that window is in focus.	fpui_open(3fpui)
		APIR3.1.1.2[31]	If application programs use graphics on a window, the API shall not redisplay these graphics when a window is refreshed or goes out/in focus.	fpui_open(3fpui)
		APIR3.1.1.2[32]	The API shall provide an asynchronous notification to alert programs when their associated windows go in and out of focus.	The Linux SIGWINCH signal is provided. Application level handler functions can assess changes.
		APIR3.1.1.2[33]	The API shall provide a function which application programs may use to determine if their window is in focus.	fpui_get_focus(3fpui)
		APIR3.1.1.2[34]	The API shall provide a method to allow application programs to indicate that a window desires focus from the Operational User.	fpui_emergency(3fpui)
		APIR3.1.1.2[35]	This method shall cause the Front Panel backlight to flash and the window name in the Front Panel Manager Window to blink.	fpui_emergency(3fpui)
		APIR3.1.1.2[36]	The window name blinking shall cease once the indicated window receives focus.	fpui_emergency(3fpui)
		APIR3.1.1.2[37]	The backlight flashing shall cease when all windows requesting focus have been given focus.	fpui_emergency(3fpui)
		APIR3.1.1.2[38]	The API shall provide a mechanism to allow application programs to detect the presence or absence of a front panel.	fpui_get_window_size(3fpui)
		APIR3.1.1.2[39]	The API shall recognize the presence or absence of the front panel in 5 seconds.	This is an operational requirement for the API software.
		APIR3.1.1.2[40]	The API shall provide an asynchronous notification to alert application programs of a change in the presence or absence of the front panel.	The Linux SIGWINCH signal is provided. Application level handler functions can assess changes.

		APIR3.1.1.2[41]	The API shall provide an asynchronous notification to alert all application programs when their associated windows change size.	The Linux SIGWINCH signal is provided. Application level handler functions can assess changes.
		APIR3.1.1.2[42]	The API shall provide a function to allow application programs to reset the display as described in the ATC Controller Standard, Section 7.1.4.	fpui_reset(3fpui)
		APIR3.1.1.2[43]	The API shall provide a function to illuminate or extinguish the CPU ACTIVE LED described in the ATC Controller Standard, Section 7 and Section B.2.	fpui_set_led(3fpui)
		APIR3.1.1.2[44]	The function shall only operate for application programs with a window in focus.	fpui_set_led(3fpui)
		APIR3.1.1.2[45]	The API shall provide a function to send raw output data to the display.	fpui_write(3fpui), fpui_open(3fpui)
		APIR3.1.1.2[46]	If the application window is in focus, the data shall be sent to the display port without interpretation or buffering by the API.	fpui_write(3fpui), fpui_open(3fpui)
		APIR3.1.1.2[47]	If the application window is not in focus, the API shall discard the data.	fpui_write(3fpui), fpui_open(3fpui)
		APIR3.1.1.2[48]	The API shall provide a function to read raw input data from the display (this does not include the Aux Switch which is handled separately; see Item “c”).	fpui_read(3fpui), fpui_open(3fpui)
		APIR3.1.1.2[49]	This function shall return raw data from the input buffer without the key code interpretation described in item “y.”	fpui_read(3fpui), fpui_open(3fpui)
APIN2.1.3[1]	These managed resources include the Front Panel (see Section 2.1.2 User Interfaces) and the Field I/O Devices.	APIR3.1.1[1]	The API shall provide a text-based user interface capability to allow application programs running concurrently on an ATC controller unit to share the controller’s Front Panel display.	fpui_open(3fpui)
		APIR3.1.2[1]	The API shall assume it has exclusive access to the serial communications ports of the ATC Engine Board that are designated for Field I/O Devices.	This is an operational requirement for the API software. The API functions in Section 4.2 are defined accordingly.
APIN2.1.3[2]	The API does not communicate directly to these ATC resources but through the Linux operating system and associated device drivers provided with the ATC controller unit.	APIR3.6[4]	The API software shall only reference operating system commands and features that are available in the Linux environment defined in the ATC Board Support Package (see ATC Controller Standard, Section 2.2.5, Annex A and Annex B).	This is a general design requirement for the API software.
APIN2.1.6[1]	The API must operate effectively within the memory constraints defined for ATC controller units in the ATC Controller Standard.	APIR3.4[1]	The API shall operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard	This is an operational requirement for the API software.

APIN2.2[1]	Operational Users must interface with application programs through the API to perform the primary tasks assigned to the controller unit.	APIR3.1.1.1[1] through APIR3.1.1.1[19]	Listed Previously	Listed Previously
APIN2.2[2]	Utility functions are necessary for configuration purposes.	APIR3.2[1]	The API shall provide a method to determine the version number(s) of the API.	fpui_apiver(3fpui), fio_apiver(3fio)
		APIR3.2[2]	The API shall provide a function to allow application programs to set the system time.	tod_set(3tod), tod_get(3tod), tod_set_dst_state(3tod), tod_get_dst_state(3tod), tod_get_dst_info(3tod), tod_set_dst_info(3tod), tod_get_timesrc(3tod), tod_set_timesrc(3tod), tod_get_timesrc_freq(3tod), tod_request_tick_signal(3tod), tod_cancel_tick_signal(3tod), tod_request_onchange_signal(3tod), tod_cancel_onchange_signal(3tod).
APIN2.2[3]	The API Manager Functions and the API Utilities access the ATC controller hardware through the Linux Kernel.	APIR3.4[1]	The API shall operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard.	This is an operational requirement for the API software.
APIN2.2[4]	Front Panel Manager. This set of functions provides a user interface which allows programs running concurrently on an ATC controller to share the controller's Front Panel through the dedicated Front Panel Port of the ATC Engine Board.	APIR3.1.1[1]	The API shall provide a text-based user interface capability to allow application programs running concurrently on an ATC controller unit to share the controller's Front Panel display.	fpui_open(3fpui)
		APIR3.1.1[2]	The API shall provide up to 16 virtual display screens (referred to as "windows") that can be used by application programs as their user interface display.	fpui_open(3fpui)
		APIR3.1.1[3]	The display size of the windows shall be equal to physical display size (lines x characters) of the controller's Front Panel display (if one exists).	fpui_open(3fpui)
		APIR3.1.1[4]	The display size of the windows shall have a minimum size of 4 lines x 40 characters and a maximum size of 24 lines x 80 characters.	fpui_open(3fpui)
		APIR3.1.1[5]	If no physical display exists, the API shall operate as if it has a display with a size of 8 lines x 40 characters.	fpui_open(3fpui)
		APIR3.1.1[6]	Only one window shall be displayed at a time on the Front Panel display.	fpui_open(3fpui)

		APIR3.1.1[7]	When a window is displayed, the API shall display the character representation of the window on the Front Panel display (if one exists).	fpui_open(3fpui)
		APIR3.1.1[8]	The application program associated with the window displayed shall receive the characters input from the Front Panel input device (Ex. keyboard or keypad).	fpui_open(3fpui)
		APIR3.1.1[9]	The API shall support the display character set as defined in the ATC Controller Standard, Section 7.1.4.	fpui_open(3fpui)
		APIR3.1.1[10]	Screen attributes described by the ATC Controller Standard, Section 7.1.4, shall be maintained for each window independently.	fpui_open(3fpui)
		APIR3.1.1[11]	Each window shall have separate input and output buffers unique from other windows.	fpui_open(3fpui)
		APIR3.1.1.2[1] through APIR3.1.1.2[49]	Listed Previously	Listed Previously
APIN2.2[5]	Field I/O Manager. This set of functions gives concurrently running programs the capability to communicate with Field I/O Devices connected to an ATC controller through the dedicated Field I/O ports of the ATC Engine Board.	APIR3.1.2[1]	The API shall assume it has exclusive access to the serial communications ports of the ATC Engine Board that are designated for Field I/O Devices.	fio_register(3fio)
		APIR3.1.2[2]	The supported Field I/O serial communications ports shall be SP3, SP5 and SP8.	fio_fiod_register(3fio), FIO_PORT parameter
		APIR3.1.2[3]	The supported communication modes on those ports shall be 153.6 Kbps and 614.4 Kbps SDLC.	fio_fiod_register(3fio)
		APIR3.1.2[4]	The API shall not open any serial communications port or initiate communications to any Field I/O Device unless explicitly commanded to do so by an application program.	fio_fiod_enable(3fio)
		APIR3.1.2[5]	The API shall support all cabinet architectures and associated Field I/O Device types as listed in the ATC Controller Standard Section 8.	fio_register(3fio)
		APIR3.1.2[6]	The API shall support the Field I/O Device types shown in Table 2.	fio_fiod_register(3fio), FIO_DEVICE_TYPE parameter
		APIR3.1.2[7]	The API shall assume that BIU and MMU Field I/O Devices operate at 153.6 Kbps and all other Field I/O Device types operate at 614.4 Kbps.	fio_fiod_register(3fio)

		APIR3.1.2[8]	The API shall support communication to multiple Field I/O Devices on a single communications port provided the Field I/O Devices have compatible physical communication attributes.	fio_fiod_register(3fio)
		APIR3.1.2[9]	The API shall support a maximum of one Field I/O Device of each type per communications port except in the case of BIUs and SIUs.	fio_fiod_register(3fio), FIO_DEVICE_TYPE parameter
		APIR3.1.2[10]	The API shall support up to 8 Detector BIU and 8 Terminal & Facilities BIU Field I/O Devices per communications port.	fio_fiod_register(3fio), FIO_DEVICE_TYPE parameter
		APIR3.1.2[11]	The API shall support up to 5 Input SIU, 2 14-Pack Output SIU and 4 6-Pack Output SIU Field I/O Devices per communications port.	fio_fiod_register(3fio), FIO_DEVICE_TYPE parameter
		APIR3.1.2[12]	The API shall only support valid Output SIU combinations as defined in the ITS Cabinet Standard, Section 4.7.	fio_fiod_register(3fio), FIO_DEVICE_TYPE parameter
		APIR3.1.2[13]	The API shall identify specific Field I/O Devices using the API Field I/O Device Names in Table 2.	FIO_DEVICE_TYPE is an enum meeting this requirement, fio_fiod_register(3fio)
		APIR3.1.2[14]	The API shall provide a method for application programs to register and deregister with the API for access to the API Field I/O services.	fio_register(3fio), fio_deregister(3fio)
		APIR3.1.2[15]	The process of application program registration shall not cause the API to perform any communications with the Field I/O Device.	fio_register(3fio)
		APIR3.1.2[16]	When an application program deregisters for access to Field I/O services, the API shall deregister (as defined in Item “e”) all Field I/O devices registered by that application program.	fio_deregister(3fio)
		APIR3.1.2[17]	The API shall provide a method to allow application programs to register and deregister for access to specific Field I/O Devices by specifying the communications port, device type, and where applicable, the Field I/O Device number.	fio_fiod_register(3fio), fio_fiod_deregister(3fio)
		APIR3.1.2[18]	Once a device has been registered on a communications port, the API shall permit the registration of additional compatible Field I/O Devices on the same communications port and prohibit the registration of incompatible Field I/O Devices on the same communications port.	fio_fiod_register(3fio)
		APIR3.1.2[19]	The Field I/O Device registration process shall not cause the API to perform any device communications.	fio_fiod_register(3fio)

		APIR3.1.2[20]	When an application program deregisters for access to a Field I/O Device, the API shall disable (as defined in Item “g”) the Field I/O Device, relinquish all output points for that device and set all application program settable states to their default values.	fio_fiod_deregister(3fio)
		APIR3.1.2[21]	The API shall provide a method for application programs to query for the presence of a Field I/O Device using the communications port, device type, and where applicable, the Field I/O Device number.	fio_query_fiod(3fio)
		APIR3.1.2[22]	If the API does not have the communications port open at the time of the query and it is necessary for the API to open the communications port to determine the Field I/O Device, the API shall close the communications port after the query is completed.	fio_query_fiod(3fio)
		APIR3.1.2[23]	If the API has the communications port open at the time of the query and the communications attributes for the Field I/O Device used in the query are not compatible with the current settings on the communications port, the API shall assume that the Field I/O Device is not present.	fio_query_fiod(3fio)
		APIR3.1.2[24]	If the API has the communications port open at the time of the query and API is already successfully completing scheduled communications to the Field I/O Device, the API shall indicate that the Field I/O Device is present without sending any additional frames to the device.	fio_query_fiod(3fio)
		APIR3.1.2[25]	The API shall provide a method which allows an application program to enable and disable communications to a Field I/O Device for which the application program has registered.	fio_fiod_enable(3fio), fio_fiod_disable(3fio)
		APIR3.1.2[26]	When the communications enable method is called, the API shall initiate scheduled communications between the API and the specified Field I/O Device if not already active.	fio_fiod_enable(3fio)
		APIR3.1.2[27]	When the disable communications method is called, the API shall cease scheduled communications between the API and the specified Field I/O Device if the device is no longer enabled by any application program.	fio_fiod_disable(3fio)
		APIR3.1.2[28]	When a Field I/O Device is disabled, any output points which have been reserved by that application program shall be set to Off.	fio_fiod_disable(3fio)

		APIR3.1.2[29]	The API shall provide a method for application programs to read the states of the input and output points on registered Field I/O Devices, including both filtered and non-filtered states for the input points (depending on which input frames are scheduled).	fio_fiod_inputs_get(3fio), fio_fiod_outputs_get(3fio)
		APIR3.1.2[30]	If multiple application programs have registered for the same Field I/O Device, the API shall provide shared read access to the input and output point states for all application programs which have registered that device.	fio_fiod_inputs_get(3fio), fio_fiod_outputs_get(3fio)
		APIR3.1.2[31]	When the state of an output point is read, the API shall return the current state of that output point within the API.	fio_fiod_outputs_get(3fio)
		APIR3.1.2[32]	The API shall provide a method for application programs to reserve/relinquish exclusive “write access” to individual output points of a Field I/O Device.	fio_fiod_outputs_reservation_set(3fio)
		APIR3.1.2[33]	If an application program reserves a point that has already been reserved by that application program, it shall not be considered an error.	fio_fiod_outputs_reservation_set(3fio)
		APIR3.1.2[34]	If an application program relinquishes a point that is already in the relinquished state for that application program, it shall not be considered an error.	fio_fiod_outputs_reservation_set(3fio)
		APIR3.1.2[35]	If a point in a group of points cannot be reserved, the reservation attempt shall fail for all of them.	fio_fiod_outputs_reservation_set(3fio)
		APIR3.1.2[36]	The API shall allow only one application program to reserve write access to any individual output point.	fio_fiod_outputs_reservation_set(3fio)
		APIR3.1.2[37]	The API shall allow multiple application programs to reserve different output points on a single Field I/O Device.	fio_fiod_outputs_reservation_set(3fio)
		APIR3.1.2[38]	Exclusive reservation of an output point for write access by one application program shall not preclude other application programs from reading the state of the output point.	fio_fiod_outputs_get(3fio)
		APIR3.1.2[39]	The API shall provide error codes so that the application program can determine if the reservation action was successful or if there was a conflict with another application program.	fio_fiod_outputs_reservation_set(3fio)
		APIR3.1.2[40]	The API shall make output point reservations on a “first come first served basis.”	fio_fiod_outputs_reservation_set(3fio)

		APIR3.1.2[41]	An application program shall be able to set the state of an output point if it has registered the associated Field I/O Device and reserved exclusive write access to the output point.	fio_fiod_outputs_set(3fio)
		APIR3.1.2[42]	To set the state of an output point and control dimming, the API shall use separate arrays for control of the Load Switch + and Load Switch – (see Section 3.3.1.4.1.5 of the TS 2 Standard).	fio_fiod_outputs_set(3fio)
		APIR3.1.2[43]	The API shall provide a method for application programs to query the reservation status of output points on registered Field I/O Devices.	fio_fiod_outputs_reservation_get(3fio)
		APIR3.1.2[44]	The API shall provide a method for application programs to map/unmap reserved output points to reserved channels and colors on a registered FIOMMU or FIOCMU device.	fio_fiod_channel_map_set(3fio) Provides both map and unmap functionality. fio_fiod_channel_map_count(3fio), fio_fiod_channel_map_get(3fio)
		APIR3.1.2[45]	The API shall use this mapping to set the contents of FIOMMU Frame 0 and FIOCMU Frames 61 and 67.	fio_fiod_channel_map_set(3fio), fio_fiod_outputs_set(3fio)
		APIR3.1.2[46]	Any channel and color not mapped to an output point shall be set to Off.	fio_fiod_channel_map_set(3fio)
		APIR3.1.2[47]	The API shall provide a method for application programs to reserve/relinquish exclusive control of individual monitored channels on the FIOMMU or FIOCMU device.	fio_fiod_channel_reservation_set(3fio) Provides both reserve and relinquish functionality. fio_fiod_channel_reservation_get(3fio)
		APIR3.1.2[48]	If an application program reserves a channel that has already been reserved by that application program, it shall not be considered an error.	fio_fiod_channel_reservation_set(3fio)
		APIR3.1.2[49]	If an application program relinquishes a channel that is already in the relinquished state for that application program, it shall not be considered an error.	fio_fiod_channel_reservation_set(3fio)
		APIR3.1.2[50]	If a channel in a group of channels cannot be reserved, the reservation attempt shall fail for all of them.	fio_fiod_channel_reservation_set(3fio)
		APIR3.1.2[51]	The API shall allow multiple applications to reserve different channels on a single FIOMMU or FIOCMU device.	fio_fiod_channel_reservation_set(3fio)
		APIR3.1.2[52]	The API shall provide error codes so that the application program can determine if the reservation action was successful or if there was a conflict with another application.	fio_fiod_channel_reservation_set(3fio)
		APIR3.1.2[53]	The API shall make channel reservations on a “first come first served basis.”	fio_fiod_channel_reservation_set(3fio)

		APIR3.1.2[54]	The API shall provide a method for applications to query the reservation status of channels on registered FIOMMU or FIOCMU devices.	fio_fiod_channel_reservation_get(3fio)
		APIR3.1.2[55]	Relinquishing a reserved output point or channel shall clear the associated assignments.	fio_fiod_outputs_reservation_set(3fio), fio_fiod_channel_reservation_set(3fio), fio_fiod_channel_map_set(3fio), fio_fiod_channel_map_count(3fio)
		APIR3.1.2[56]	The API shall provide functions which allow application programs to set and get the leading and trailing edge filter values on a per input basis for all Field I/O Devices that support configurable filtered inputs.	fio_fiod_inputs_filter_set(3fio), fio_fiod_inputs_filter_get(3fio)
		APIR3.1.2[57]	If multiple application programs set the filter values of an input, the shortest filter values shall be used.	fio_fiod_inputs_filter_set(3fio)
		APIR3.1.2[58]	The API shall provide a return code containing the status and the value used for the set filter operation.	fio_fiod_inputs_filter_set(3fio)
		APIR3.1.2[59]	The default leading and trailing edge filter values shall be 5 consecutive samples.	#define constant FIO_FILTER_DEFAULT 5, fio_fiod_inputs_filter_set(3fio)
		APIR3.1.2[60]	The API shall have the ability to collect and buffer the transition buffer information for each registered Field I/O Device used for input.	fio_fiod_inputs_trans_set(3fio), fio_fiod_inputs_trans_read(3fio)
		APIR3.1.2[61]	When the API reads the transition buffer of a Field I/O Device, it shall read the entire transition buffer.	fio_fiod_inputs_trans_read(3fio)
		APIR3.1.2[62]	The API shall buffer the transition data on a per application program basis with the capability of storing 1024 transition entries in a FIFO fashion.	fio_fiod_inputs_trans_read(3fio)
		APIR3.1.2[63]	The API shall provide a function which allows application programs to enable or disable transition monitoring of selected input points.	fio_fiod_inputs_trans_set(3fio), fio_fiod_inputs_trans_get(3fio)
		APIR3.1.2[64]	By default, transition monitoring for all input points shall be disabled.	fio_fiod_inputs_trans_set(3fio)
		APIR3.1.2[65]	If an application program enables an input point for transition monitoring and that input point is already in the enabled state, it shall not be considered an error.	fio_fiod_inputs_trans_set(3fio)
		APIR3.1.2[66]	If an application program disables an input point for transition monitoring and that input point is already in the disabled state, it shall not be considered an error.	fio_fiod_inputs_trans_set(3fio)

		APIR3.1.2[67]	The API shall provide functions that allow application programs to access the API transition buffer information asynchronously (i.e. read the transition entries from the API buffer independent of any Field I/O Device communications).	fio_fiod_inputs_trans_read(3fio)
		APIR3.1.2[68]	When an application program reads a transition entry from an API transition buffer, that transition entry shall be cleared for that application program only, without affecting the API transition buffers for other application programs.	fio_fiod_inputs_trans_read(3fio)
		APIR3.1.2[69]	If the transition buffer in the Field I/O Device overruns before information can be copied to the API transition buffer information, the API shall indicate that a device overrun condition has occurred in the transition buffer for that Field I/O Device.	fio_fiod_inputs_trans_read(3fio) The return value is success (count of entries), FIOD Overrun or FIO API overrun.
		APIR3.1.2[70]	If the transition buffer of the API overruns before the information is retrieved by the application program, the API shall indicate that an API overrun condition has occurred.	fio_fiod_inputs_trans_read(3fio) The return value is success (count of entries), FIOD Overrun or FIO API overrun.
		APIR3.1.2[71]	The ATC Controller Standard, Section 8, specifies the frames for communication with Field I/O Devices for Model 332 Cabinets, NEMA TS 1 and TS 2 Type 2 Cabinets and ITS Cabinets. The API shall support a subset of these frames at the scheduled frame frequencies as shown in Table 3.	fio_fiod_frame_schedule_set(3fio)
		APIR3.1.2[72]	The NEMA TS 2 Standard, Section 3.3, specifies the frames for communication with Field I/O Devices for NEMA TS 2 Type 1 Cabinets. The API shall support a subset of these frames at the scheduled frame frequencies as shown in Table 4.	fio_fiod_frame_schedule_set(3fio)
		APIR3.1.2[73]	The timing for the command/response cycle of the frames shall be defined by the “Handshaking” algorithm in Section 3.3.1.5.3 of the NEMA TS 2 Standard.	fio_fiod_frame_schedule_set(3fio)
		APIR3.1.2[74]	The API shall provide a method for application programs to set/get the scheduled frame frequencies for a registered Field I/O Device.	fio_fiod_frame_schedule_set(3fio), fio_fiod_frame_schedule_get(3fio)
		APIR3.1.2[75]	The frame frequency used by the API shall be the highest frequency requested by all application programs registered for that Field I/O Device.	fio_fiod_frame_schedule_set(3fio)

		APIR3.1.2[76]	The API shall provide a method to send a frame from either Table 3 or Table 4 one time (non-scheduled).	fio_fiod_frame_schedule_set(3fio) frequency of FIO_HZ_ONCE
		APIR3.1.2[77]	The API shall provide a method to set/get the Failed State Action of a FIOCMU Field I/O Device.	fio_fiod_cmu_fault_set(3fio), fio_fiod_cmu_fault_get(3fio)
		APIR3.1.2[78]	The Failed State Action shall be settable to None (LFSA=0, NFSA=0), Latched (LFSA=1, NFSA=0), or Non Latched (LFSA=0, NFSA=1).	fio_fiod_cmu_fault_set(3fio)
		APIR3.1.2[79]	The default Failed State Action shall be None.	fio_fiod_cmu_fault_set(3fio)
		APIR3.1.2[80]	If any application program sets the state to Latched, the API shall set the Failed State Action to Latched.	fio_fiod_cmu_fault_set(3fio)
		APIR3.1.2[81]	If no application program has set the Failed State Action to Latched, then if any application program sets the state to Non Latched, the API shall set the Failed State Action to Non Latched.	fio_fiod_cmu_fault_set(3fio)
		APIR3.1.2[82]	If all application programs have a state of None, then the API shall set the Failed State Action to None.	fio_fiod_cmu_fault_set(3fio)
		APIR3.1.2[83]	The API shall provide a method to set/get the state of the Fault Monitor output point of FIOTS1 and FIOTS2 Field I/O Devices.	fio_fiod_ts_fault_monitor_set(3fio), fio_fiod_ts_fault_monitor_get(3fio)
		APIR3.1.2[84]	The API shall retain ownership of the Fault Monitor output point and not allow application programs to reserve this output point.	fio_fiod_ts_fault_monitor_set(3fio)
		APIR3.1.2[85]	If any application program sets the Fault Monitor state to Off, the API shall turn Off the Fault Monitor output point on that device.	fio_fiod_ts_fault_monitor_set(3fio)
		APIR3.1.2[86]	If all application programs have a Fault Monitor state of On for a FIOTS1 or FIOTS2 Device, then the API shall turn On the Fault Monitor output point on that device.	fio_fiod_ts_fault_monitor_set(3fio)
		APIR3.1.2[87]	The default state of the Fault Monitor output point shall be On.	fio_fiod_ts_fault_monitor_set(3fio)
		APIR3.1.2[88]	The API shall provide a method to set/get the state of the Voltage Monitor output point of a FIOTS1 Field I/O Device.	fio_fiod_ts1_volt_monitor_set(3fio), fio_fiod_ts1_volt_monitor_get(3fio)
		APIR3.1.2[89]	The API shall retain ownership of the Voltage Monitor output point and not allow application programs to reserve this output point.	fio_fiod_ts1_volt_monitor_set(3fio)
		APIR3.1.2[90]	If any application program sets the Voltage Monitor state to Off, the API shall turn Off the Voltage Monitor output point on that device.	fio_fiod_ts1_volt_monitor_set(3fio)

		APIR3.1.2[91]	If all application programs have a Voltage Monitor state of On for a FIOTS1 Device, then the API shall turn On the Voltage Monitor output point on that device.	fio_fiod_ts1_volt_monitor_set(3fio)
		APIR3.1.2[92]	The default state of the Voltage Monitor output point shall be On.	fio_fiod_ts1_volt_monitor_set(3fio)
		APIR3.1.2[93]	The API shall provide a method which allows application programs to assign the output point used for the Watchdog output of any registered Field I/O Device.	fio_fiod_wd_reservation_set(3fio), fio_fiod_wd_reservation_get(3fio)
		APIR3.1.2[94]	The API shall restrict the ability to assign the Watchdog output point to the first application program to call the assignment method.	fio_fiod_wd_reservation_set(3fio), fio_fiod_wd_register(3fio)
		APIR3.1.2[95]	The API shall retain ownership of the Watchdog output point and not allow application programs to reserve that output point directly.	fio_fiod_outputs_reservation_set(3fio), fio_fiod_wd_reservation_set(3fio), fio_fiod_wd_register(3fio)
		APIR3.1.2[96]	The API shall provide a method for application programs to register for shared control of the Watchdog output point.	fio_fiod_wd_register(3fio)' fio_fiod_wd_deregister
		APIR3.1.2[97]	The API shall provide a method for Watchdog registered application programs to "request" that the API toggle the state of the Watchdog output point.	fio_fiod_wd_heartbeat(3fio)
		APIR3.1.2[98]	The API shall only toggle the Watchdog output point if all Watchdog registered application programs have made the toggle request (Watchdog Triggered Condition).	fio_fiod_wd_heartbeat(3fio)
		APIR3.1.2[99]	Upon a Watchdog Triggered Condition, the API shall toggle the state of the Watchdog output point within the API.	fio_fiod_wd_heartbeat(3fio)
		APIR3.1.2[100]	When the API updates the output states of the Field I/O Device (see Item "n"), the API shall clear all previous toggle requests and the Watchdog Triggered Condition so that a new Watchdog Triggered Condition can be generated.	fio_fiod_wd_heartbeat(3fio)
		APIR3.1.2[101]	The API shall not toggle the Watchdog output point more than once per update of the output states on the Field I/O Device.	fio_fiod_wd_heartbeat(3fio)
		APIR3.1.2[102]	The API shall provide functions which allow application programs to obtain status information of a registered Field I/O Device.	fio_fiod_status_get(3fio)
		APIR3.1.2[103]	All counters contained in the Field I/O Device status information shall be four byte unsigned values each with a maximum value of 4,294,967,295.	fio_fiod_status_get(3fio)

		APIR3.1.2[104]	The counters shall be frozen when they reach the maximum value to prevent rollover.	fio_fiod_status_get(3fio)
		APIR3.1.2[105]	The API shall provide the following communication status information for each registered Field I/O Device: i) Communications Enabled/Disabled; ii) Cumulative successful response count for all frames to this device; iii) Cumulative error count for all frames to this device; and iv) Command frames sent to this device with the following information for each frame type: current scheduled frequency, cumulative successful response count, cumulative error count, numbers of errors in the last 10 frames, a response frame sequence number, frame size in bytes and the raw data from the most recent response frame.	fio_fiod_status_get(3fio) Frame access supported by fio_fiod_frame_read(3fio). Frame size and sequence number are supported by fio_fiod_frame_size(3fio). A sequence number will be provided on all returned frames for frame aging.
		APIR3.1.2[106]	The response frame sequence number shall be a four byte unsigned value and rollover after the maximum value.	fio_fiod_frame_size(3fio), fio_fiod_frame_read(3fio), fio_query_frame_notify_status(3fio)
		APIR3.1.2[107]	The API shall provide a method for application programs to reset the communications status counters to 0 (zero) for a registered Field I/O Device.	fio_fiod_status_reset(3fio)
		APIR3.1.2[108]	A response frame shall only be considered successful if it is fully received within the time period defined by the “Handshaking” algorithm in Section 3.3.1.5.3 of the NEMA TS 2 Standard.	fio_fiod_status_get(3fio)
		APIR3.1.2[109]	The API shall provide an API Health Monitor Function which registered application programs use to indicate to the API that they are operational.	fio_hm_heartbeat(3fio)
		APIR3.1.2[110]	The API shall provide a method to set an API Health Monitor Timeout for each application program (each application program has its own unique API Health Monitor Timeout).	fio_hm_register(3fio), fio_hm_deregister(3fio)
		APIR3.1.2[111]	This API Health Monitor Timeout shall indicate the maximum allowable time between calls to the API Health Monitor Function.	fio_hm_register(3fio)
		APIR3.1.2[112]	The API Health Monitor Timeout shall be specified in tenths of a second.	fio_hm_register(3fio)
		APIR3.1.2[113]	If the API Health Monitor Timeout expires for an application, the API shall disable (as defined previously in Item “g”) all Field I/O Devices registered by that application program.	fio_hm_register(3fio), fio_hm_heartbeat(3fio)

		APIR3.1.2[114]	The API shall provide a method for an application program to disable the API Health Monitor feature for itself.	fio_hm_register(3fio), fio_hm_deregister(3fio)
		APIR3.1.2[115]	The API shall provide a method for an application program to reset an API Health Monitor fault condition and allow the API to resume Field I/O Device communications.	fio_hm_fault_reset(3fio), fio_hm_heartbeat(3fio)
		APIR3.1.2[116]	An application shall only be able to reset its own Health Monitor fault condition and not that of any other application program.	fio_hm_fault_reset(3fio)
		APIR3.1.2[117]	If an application program resets the API Health Monitor fault condition, then any devices that were disabled due to that condition shall be re-enabled.	fio_hm_fault_reset(3fio)
		APIR3.1.2[118]	If an application program attempts to enable a device (as defined in Item “g”) that has been disabled due to an API Health Monitor fault condition, then the enable operation shall return an error and the Field I/O Device remain disabled.	fio_hm_fault_reset(3fio), fio_fiod_enable(3fio)
		APIR3.1.2[119]	A call to the API Health Monitor Function after a Health Monitor fault has occurred shall not reset the Health Monitor fault condition.	fio_hm_fault_reset(3fio), fio_hm_heartbeat(3fio)
		APIR3.1.2[120]	The API Health Monitor Function shall return whether an API Health Monitor fault condition exists.	fio_hm_heartbeat(3fio)
		APIR3.1.2[121]	The API shall provide a method for an application program to send the Get CMU Configuration frame to a registered FIOCMU device.	This requirement is satisfied by: - Set frame notification for response frame 193 using fio_fiod_frame_notify_register(3fio) - Send frame 65 one time using fio_fiod_frame_schedule_set(3fio) - Received response frame notification: SIGIO and fio_query_frame_notify_status(3fio) - Read response frame 193 using fio_fiod_frame_read(3fio)
		APIR3.1.2[122]	The API shall reset all Module Status bits using the Request Module Status frame when a FIO332, FIOTS1, FIOTS2 or SIU device is first Enabled (as defined in Item “g”).	fio_fiod_enable(3fio), fio_fiod_frame_schedule_set(3fio)
		APIR3.1.2[123]	Anytime a response to a Request Module Status frame has Module Status bits indicating hardware reset, comm loss, or watchdog reset, then the API shall clear those bits, reset the input point filter values (Item “k”) and reconfigure transition reporting (Item “l”).	fio_fiod_enable(3fio), fio_fiod_frame_schedule_set(3fio)

		APIR3.1.2[124]	The API shall provide a method to notify an application program when a command frame is acknowledged (response frame received by the API) or when an error occurs.	fio_fiod_frame_notify_register(3fio) Allows application to say I want to be notified when this frame is received. fio_fiod_frame_notify_deregister(3fio) Allows application to dismiss the notification service. fio_query_frame_notify_status(3fio) For retrieving information on frame status after notify – what frame is being notified and why (received, error, other). This function returns the FIO_DEV_HANDLE that represents the FIOD that responded.
		APIR3.1.2[125]	The command frame shall be identified by the frame type and a registered Field I/O Device.	fio_fiod_frame_notify_register(3fio), fio_fiod_frame_notify_deregister(3fio)
		APIR3.1.2[126]	The response frame notification shall include the Field I/O Device, response frame type, response frame sequence number, response frame size in bytes and an indication as to why the notification occurred (response received or error detected).	fio_query_frame_notify_status(3fio)
		APIR3.1.2[127]	The notification shall be able to be set for a one time occurrence or continuous occurrence.	fio_fiod_frame_notify_register(3fio), fio_fiod_frame_notify_deregister(3fio)
		APIR3.1.2[128]	The API shall provide a method to set and get the Dark Channel Map selection for a registered FIOCMU device.	fio_fiod_cmu_dark_channel_set(3fio), fio_fiod_cmu_dark_channel_get(3fio)
		APIR3.1.2[129]	If multiple application programs attempt to set the Dark Channel Map selection, the API shall use the most recent selection.	fio_fiod_cmu_dark_channel_set(3fio)
		APIR3.1.2[130]	The default value of the Dark Channel Map Select bits shall be 0 (Mask #1).	fio_fiod_cmu_dark_channel_set(3fio)
		APIR3.1.2[131]	The API shall provide a method to set and get the state of the Load Switch Flash bit of a registered FIOMMU device.	fio_fiod_mmu_flash_bit_set(3fio), fio_fiod_mmu_flash_bit_get(3fio). Bit 112 of MMU frame 0. Default to value 0.
		APIR3.1.2[132]	If multiple application programs attempt to set the state of the Load Switch Flash bit, the API shall use the most recent state.	fio_fiod_mmu_flash_bit_set(3fio)
		APIR3.1.2[133]	The default value of the Load Switch Flash bit shall be 0.	fio_fiod_mmu_flash_bit_set(3fio)
		APIR3.1.2[134]	When an application program exits or terminates for any reason, the API shall deregister the application program from the API (as defined in Item “d”).	fio_deregister(3fio)

APIN2.2[6]	Functions in the area API Utilities provide the Operational User tools in which to configure and maintain the API installed on an ATC. While some of these functions are based on the implementation of the managers and outside of this standard, there is a need for Operational Users to have access to the API version and configuration information	APIR3.2[1]	The API shall provide a method to determine the version number(s) of the API.	fpui_apiver(3fpui), fio_apiver(3fio)
		APIR3.2[2]	The API shall provide a function to allow application programs to set the system time.	tod_set(3tod), tod_get(3tod), tod_set_dst_state(3tod), tod_get_dst_state(3tod), tod_get_dst_info(3tod), tod_set_dst_info(3tod), tod_get_timesrc(3tod), tod_set_timesrc(3tod), tod_get_timesrc_freq(3tod), tod_request_tick_signal(3tod), tod_cancel_tick_signal(3tod), tod_request_onchange_signal(3tod), tod_cancel_onchange_signal(3tod).
		APIR3.2.1[1]	The API shall provide a window called the ATC Configuration Window.	fpui_open(3fpui)
		APIR3.2.1[2]	Operational Users shall be able to view ATC configuration information on this window provided by the Linux uname() function (ATC Controller Standard, Section 2.2.5), the API's version number(s), and the ATC Host Module EEPROM information (ATC Controller Standard, Annex B).	fpui_open(3fpui)
		APIR3.2.1[3]	The default ATC Configuration Window size shall be 8 lines x 40 characters with the format as show in Figure 8.	fpui_open(3fpui)
		APIR3.2.1[4]	The top two lines and bottom line of the ATC Configuration Window shall be fixed as shown in Figure 8.	This is an operational requirement for the API software.
		APIR3.2.1[5]	The number of lines between the second line and bottom lines used for displaying the ATC configuration information shall vary according to the size of the ATC display.	This is an operational requirement for the API software.
		APIR3.2.1[6]	The Operational User shall be able to scroll up and down the ATC Configuration Window one line at a time to view the configuration information using the up and down arrow keys of the controller keypad.	This is an operational requirement for the API software.

		APIR3.2.1[7]	The Operational User shall be able to put the Front Panel Manager in focus by pressing {<NEXT>} in the ATC Configuration Window.	This is an operational requirement for the API software.
APIN2.4[1]	The API must operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard.	APIR3.4[1]	The API shall operate on an ATC controller unit under the hardware limitations defined in the ATC Controller Standard.	This is an operational requirement for the API software.
APIN2.4[2]	The API function calls must be specified using the C programming language as described by “ISO/IEC 9899:1999,” commonly referred to as the C99 Standard.	APIR3.4[2]	The API function calls shall be specified using the C programming language as described by “ISO/IEC 9899:1999,” commonly referred to as the C99 Standard.	Section 4 describes the API accordingly.
APIN2.4[3]	The operational look and feel of user interfaces developed for the API should be consistent with each other.	APIR3.5.2[1]	The operational look and feel of user interfaces developed for the API shall have consistent window titling conventions, scrolling methods, menu styles and selection methods.	This is an operational requirement for the API software.
APIN2.4[4]	If API functions have a similar operation to existing Linux functions, they should have a similar name and argument style to those functions to the extent possible without causing compilation issues.	APIR3.5.2[2]	If API functions have a similar operation to existing Linux functions, they shall have a similar name and argument style to those functions to the extent possible without causing compilation issues.	See fpui_close(3fpui) & close(2), fpui_open(3fpui) & open(2), fpui_read(3fpui) & read(2), fpui_write(3fpui) & write(2)
APIN2.4[5]	The API functions should use consistent naming conventions, argument styles and return values.	APIR3.5.2[3]	The API function names shall be lower case.	Section 4 defines the API functions in lower case.
		APIR3.5.2[4]	API functions shall use the Linux “errno” error notification mechanism if an error indication is expected for a function.	Section 4 API functions are defined accordingly.
APIN2.4[6]	The API should be available to users as a library that is loadable via a start-up script.	APIR3.5.2[5]	The API shall be loadable as an ELF (Executable and Linking Format) library.	This is an operational requirement for the API software.

Appendix B: Code Examples

Example software using the Front Panel:

```
#include "fpui.h"

int main( int argc, char * argv[] )
{
   fpui_handle hdl = fpui_open( O_RDWR, "Hello World" );
   fpui_clear( hdl );
   fpui_write_string( hdl, "Hello World" );
   fpui_close( hd );
}


#include "fpui.h"

int main( int argc, char * argv[] )
{
   fpui_handle hdl = fpui_open( O_RDWR, "Hello World" );
   fpui_handle dhdl = fpui_open( O_RDWR | O_DIRECT, "Hello World" );
   fpui_clear( hdl );
   while( ! fpui_get_focus( hdl ) ) {
      sleep( 0 );
   }
   fpui_write( dhdl, graphics_buffer, graphics_buffer_size );
   fpui_close( dhdl );
   fpui_close( hdl );
}
```

Example software using the Field I/O Devices…

```
#include   “fio.h”

{
/* Allocate Buffers to control inputs & outputs */
unsigned char inputs_list[ FIO_INPUT_POINTS_BYTES ];
unsigned char outputs_list[ FIO_OUTPUT_POINTS_BYTES ];
unsigned char reserve_list[ FIO_OUTPUT_POINTS_BYTES ];

/* Register with FIO API */
FIO_APP_HANDLE fh = fio_register();

/* Register DEV 1 */
FIO_DEV_HANDLE dev = fio_fiod_register( fh, FIO_SP3, FIO332 );

/* Reserve Output Points */
FIO_BITS_CLEAR( reserve_list, sizeof( reserve_list ) );
FIO_BIT_SET( reserve_list, OUTPUT_BIT_0 );  /* Reserve only output point 0 */
err = fio_fiod_outputs_reservation_set( fh, dev, reserve_list );

/* Enable Communications with the FIOD */
err = fio_fiod_enable( fh, dev );  /* Turn on I/O to DEV 1 * /

/* Start processing */
FIO_BITS_CLEAR( outputs_list, sizeof( outputs_list ) );
do
{
    sleep(1);  /* Wait for a while */
    err = fio_fiod_inputs_get( fh, dev, FIO_INPUTS_FILTERED, inputs_list );  /* Get Inputs */
    if ( FIO_BIT_TEST( inputs_list, INPUT_BIT_2 ) )  /* If something happened */
    {
        FIO_BIT_SET( outputs_list, OUTPUT_BIT_0 );  /* Turn on light */
    }
```

```
    else
    {
        FIO_BIT_CLEAR( outputs_list, OUTPUT_BIT_0 );  /* Turn off light */
    }
    err = fio_fiod_outputs_set( fh, dev, outputs_list );  /* Set the light to the state indicated
*/
}
while( 1 );  /* Do forever */

endloop:
        fio_deregister( fh );  /* cleanup all services */
}
```

Appendix C: Standard Directory Structure and File Naming Conventions

The following directory structure and file naming conventions are recommended by the API Working Group.

1) The API libraries should be stored in the “/usr/lib” directory using the names “libfpui.so” for the Front Panel Manager library, “libfio.so” for the Field I/O Manager library and “libtod.so” for the Time of Day Library.

2) Application software residing on the ATC controller unit should be stored in a subdirectory of the “/opt” directory using the company name of the vendor. For example: Software produced by “SoftwareRUs” would be stored in “/opt/SoftwareRUs” (or a further subdirectories below if desired). The executables for the application software should be symbolically linked to the “/usr/bin” directory.

Appendix D: API Reference Implementation

As a part of the development of the API Standard, portions of the standard were implemented as validation. The software was developed using the C programming language and documented using IEEE Std 1016-1998 and GNU Coding Standards.

It is the desire of the API Working Group that this software be completed as a reference implementation of the API Standard and maintained in an open source fashion for the transportation industry. To accomplish this, the API Working Group makes the recommendations listed below.

1) An API project administrator should be established to oversee the continuing development of the API software and maintain its integrity.

2) An open source project Web service be selected that includes the following:
   a) a secure server to provide a common place to store the API software;
   b) a registration capability for authorized users, source code management tools;
   c) online bug/issue tracking capabilities; and
   d) collaboration tools including e-mail lists, Web-based forums, etc.

3) License the API software using a General Public License (GPL).

INDEX