

---

# **Sage Programming Reference**

**For Maestro Access Control Systems**

Copyright © Cansec Systems Ltd. 1998-2007

# Table of Contents

<b>GETTING STARTED WITH SAGE</b> .....	<b>6</b>
What is Sage.....	7
What's New.....	8
The Sage Toolkit.....	11
Components of a Sage Application.....	13
Application Guidelines.....	14
Where to Begin.....	15
<b>SAGE TUTORIAL</b> .....	<b>16</b>
Tutorial Requirements.....	17
Initializing Sage.....	18
Sage Initialization.....	19
Establishing a Connection.....	20
Querying Available Devices.....	21
Enumerating Defined Card Readers.....	22
Enumerating Defined Input and Output Points.....	24
Receiving Maestro Events.....	26
The Sage Event Queue.....	27
Event Types.....	30
A Sample Alarm Monitor Application.....	31
Querying Connection and Device Status.....	34
Client Connection Status.....	35
Device Status.....	36
Querying Other Objects.....	37
SGLISTDEF Structure.....	38
Controlling Card Readers.....	39
Card Reader Commands.....	40
A Sample Reader Command Application.....	42
Controlling Inputs and Outputs.....	45
Input Point Commands.....	46
Output Point Commands.....	47
A Sample IO Command Application.....	48
Working with Groups.....	52
Card and Device Group Functions.....	53
A Sample Group Application.....	54
Controlling Card Reader Access.....	56
Access Profiles.....	57
A Sample Access Profile Application.....	59
The Personnel Database.....	61
Personnel Database Functions.....	62
A Sample Personnel Database Application.....	65
Retrieving Audit Trail Information.....	67
Audit Trail Commands.....	68
A Sample Audit Trail Application.....	69
<b>THE SAGE API REFERENCE</b> .....	<b>71</b>
Initialization Functions.....	72
SgInitialize.....	73

SgOpenSession .....	74
SgCloseSession.....	75
SgGetWorkingDirectory .....	76
SgGetRunTimeRecord.....	77
SgGetDefinedReaders.....	78
SgGetDefinedInputs.....	79
SgGetDefinedOutputs.....	80
SgGetLocalName.....	81
Events .....	82
Message Structure and Event Types.....	83
Message Structure .....	84
Event Types .....	87
Access Denied .....	88
Access Granted .....	92
Card Reader Events.....	95
IO Point Events .....	98
Programming Events.....	100
Miscellaneous Events .....	110
Event Functions .....	111
SgGetMsg.....	112
SgSetMsgMask.....	113
Status Functions .....	114
SgGetConnectionStatus .....	115
SgGetReaderStatus.....	116
SgGetReaderStatusList .....	117
SgGetIOPointStatus.....	118
SgGetInputStatusList.....	119
SgGetOutputStatusList .....	120
Login Functions.....	121
SgLogin .....	122
SgLogout.....	123
Reader Configuration Functions .....	124
Reader Configuration Definitions and Structures .....	125
SgLoadReaderConfig .....	128
SgSaveReaderConfig .....	129
Card Reader Functions.....	130
SgUnlockReader .....	131
SgUnlockMReader.....	132
SgRelockReader .....	133
SgLockoutReader .....	134
SgRemoveLockoutReader.....	135
SgLockElevatorReader .....	136
SgUnlockElevatorReader.....	137
Input Point Functions .....	138
SgShuntInput .....	139
SgShuntMInput .....	140
SgUnshuntInput .....	141
Output Point Functions.....	142
SgActivateOutput .....	143
SgActivateMOutput .....	144
SgDeactivateOutput.....	145
Access Profile Functions.....	146
Access Profile Structure.....	147
SgGetAccessProfile .....	148
SgAddAccessProfile .....	149
SgUpdateAccessProfile .....	150

SgDeleteAccessProfile .....	151
SgOpenProfileDatabase .....	152
SgCloseProfileDatabase .....	153
SgGetFirstAccessProfile .....	154
SgGetNextAccessProfile .....	155
SgGetAccessProfileCount .....	156
SgGetDefinedAccessProfiles .....	157
Personnel Database Functions .....	158
Personnel Record Structure .....	159
SgInitCardRecord .....	162
SgGetCardRecord .....	163
SgUpdateCardRecord .....	164
SgAddCardRecord .....	165
SgVoidCardRecord .....	166
SgDeleteCardRecord .....	167
SgOpenCardDatabase .....	168
SgCloseCardDatabase .....	169
SgGetFirstCardRecord .....	170
SgGetNextCardRecord .....	171
SgFindCardRecord .....	172
SgGetCardRecordCount .....	173
SgGetDefinedCards .....	174
SgGetPersonnelTitles .....	175
SgPutPersonnelTitles .....	176
Group Functions .....	177
Group Definitions and Structures .....	178
SgGroupAdd .....	179
SgGroupSave .....	180
SgGroupDelete .....	181
SgGroupLoad .....	182
SgGetDefinedGroups .....	183
Audit Trail Functions .....	184
SgOpenAuditTrail .....	185
SgCloseAuditTrail .....	186
SgReadAuditTrail .....	187
Schedule Functions .....	188
Schedule Structure .....	189
SgGetAccessSchedule .....	190
SgUpdateAccessSchedule .....	191
SgGetUnlockSchedule .....	192
SgUpdateUnlockSchedule .....	193
SgGetElevUnlockSchedule .....	194
SgUpdateElevUnlockSchedule .....	195
SgGetShuntSchedule .....	196
SgUpdateShuntSchedule .....	197
SgGetOutputSchedule .....	198
SgUpdateOutputSchedule .....	199
SgGetSystemSchedule .....	200
SgUpdateSystemSchedule .....	201
Control Profile Functions .....	202
Control Profile Structure .....	203
SgAddCtrlProfile .....	206
SgGetCtrlProfile .....	207
SgUpdateCtrlProfile .....	208
SgDeleteCtrlProfile .....	209

SgGetDefinedCtrlProfiles .....	210
Association Functions .....	211
Association Definitions and Structures .....	212
SgLoadAssociations .....	213
SgSaveAssociations .....	214
Miscellaneous Functions .....	215
SgGetDefinedMacros .....	216
SgLogError .....	217
SgIsSysTZone .....	218
SgGlobalAPB .....	219
SgGlobalAPBState .....	220
SgDateTime .....	221
SgRestoreComms .....	222

# Getting Started with Sage

## What is Sage

Sage is a set of API's which permits independent development of software which can extend the functionality of the base Maestro access control system. Sage applications connect to a Maestro server via a WIN32 named pipe. Software may be developed which can monitor and act upon field device events, access and modify system database files, and program field control panels. Sage will allow seamless integration of third party developed software without the risk of reverse engineered code.

The Sage Developers Kit provides sample applications with source code demonstrating field device control, monitoring, and database access. The intent is to demonstrate the use of the API calls and help minimize development time. The Sage API will only support the Maestro WIN32 platform. Cansec's MS-DOS and OS/2 platforms are not supported in this manner.

# What's New

## New For Version 2.0

Version 2.0 of the Sage toolkit adds several new API calls and has optimized the performance of some of the others. New API calls include the following.

### Initialization

- SgSetMsgMask
- SgGetRunTimeRecord
- SgGetWorkingDirectory

### Personnel Database

- SgInitCardRecord
- SgOpenCardDatabase
- SgCloseCardDatabase
- SgGetPersonnelTitles
- SgPutPersonnelTitles
- SgFindCardRecord

### Access Profile Database

- SgOpenProfileDatabase
- SgCloseProfileDatabase
- SgGetFirstAccessProfile
- SgGetNextAccessProfile
- SgGetAccessProfileCount

### Miscellaneous

- SgLogMsg
- SgGetLocalName
- SgIsSysTZone

API calls that have changed include the following.

- SgGetFirstCardRecord
- SgGetNextCardRecord

In addition, the structure of the Maestro event messages has changed to return additional card holder and programming related information. See Message Structure for a description of the data fields.

## New For Version 2.1

Enhancements have been made to increase the throughput of Maestro and Sage based clients over the network. For this reason, a version 2.1 Sage.dll must be used when connecting with a Maestro server.

Version 2.1 of Sage adds support for Maestro Patient Wandering with the new card status *Patient* (see Personnel Record Structure), and the new transactions PATIENT\_WANDER and MULTI\_PATIENT\_WANDER (see Access Denied).

### **New For Version 3.0**

Version 3.0 of Sage continues to enhance API functionality through the addition of many new calls, transparent support for Maestro Partitioning, and logon security. Applications must be re-linked with this new version of the API in order to be compatible with Maestro Version 3.0. New API calls include the following.

#### Schedules

- SgGetAccessSchedule
- SgUpdateAccessSchedule
- SgGetUnlockSchedule
- SgUpdateUnlockSchedule
- SgGetElevUnlockSchedule
- SgUpdateElevUnlockSchedule
- SgGetShuntSchedule
- SgUpdateShuntSchedule
- SgGetOutputSchedule
- SgUpdateOutputSchedule
- SgGetSystemSchedule
- SgUpdateSystemSchedule

#### Control Profiles

- SgGetCtrlProfile
- SgAddCtrlProfile
- SgUpdateCtrlProfile
- SgDeleteCtrlProfile
- SgGetDefinedCtrlProfiles

#### Enumeration

- SgGetDefinedGroups
- SgGetDefinedAccessProfiles
- SgGetDefinedCards
- SgGetDefinedMacros

#### Status

- SgGetReaderStatusList
- SgGetInputStatusList
- SgGetOutputStatusList

#### Associations

- SgLoadAssociations

SgSaveAssociations

Reader Configuration

SgLoadReaderConfig

SgSaveReaderConfig

Login

SgLogin

SgLogout

Miscellaneous

SgGlobalAPB

SgGlobalAPBState

SgDateTime

SgRestoreComms

**New For Version 3.1**

The new event mask type SG\_TYPE\_DC\_EVENTS was added to the SgSetMsgMask set of mask types.

**New For Version 3.2**

There are no changes for this release of Sage.

# The Sage Toolkit

Version 3.2 of the Sage toolkit consists of the following subdirectories and files. The *Maestro* directory contains a Maestro Network installation. The following files are present;

## **FileUtil.exe**

This program file is the Maestro file utility program. It performs data file backup and restore as well as database re-indexing and file repair.

## **Maesrvr.exe**

This is the Maestro server program. This program communicates with the field control panels as well as Maestro and Sage client applications.

## **Maestro.exe**

This is the Maestro client program, which provides the main user interface for the system.

## **Upgrade.exe**

A utility to verify data file revisions and upgrade them as required.

## **Winmc.exe**

This is the Maestro macro compiler. Maestro supports a compiled macro language for customized control of field devices.

## **Wssetup.exe**

This is the Maestro client setup program. This program is run from Windows client computers to setup up the client software.

## **MFC42.dll and MSVCRT.dll**

These files are Microsoft runtime DLL's used by Maestro.

## **Maestro.cnt and Maestro.hlp**

These are the Maestro contents and help files respectively.

The *ToolKit* directory contains the files necessary for creating a Sage application. The following files are present;

## **Sage.dll**

This dynamic link library contains the Sage API runtime code.

## **Sage.lib**

This lib contains the compile/link time Sage API entry points. This file will be linked with, when the application is created.

## **Sage.h**

Sage.h contains prototypes and definitions for the Sage API. This file will be included with your source files.

**Sage.hlp** and **Sage.cnt**

These files contain the API tutorial and reference.

**Msvcrt.dll**

This dynamic link library contains the Microsoft Visual C runtime routines which Sage uses during execution.

The *Samples* subdirectory contains five Microsoft Visual C++ 4.0, console based projects. Each of these applications is referenced in the tutorial section of this on-line help guide. The following subdirectories and projects are present.

**AlrmMon**

This project demonstrates a Sage based monitor, which watches for certain events and announces them on screen.

**AudView**

This project demonstrates opening and retrieving records from the Maestro audit trail.

**Card**

This project prompts for card numbers, retrieves, and displays personnel records from the personnel database.

**IOPoint**

This project commands output points to activate and de-activate and input points to shunt and unshunt.

**Reader**

This project commands card readers to lock and unlock.

## **Components of a Sage Application**

The bare minimum requirements for a Sage application would be a single CP30/40 control panel and a Maestro server. The Maestro server would provide back end communications to the control panel and the named pipe communications channel to the Sage application. In practical terms though a Maestro client would be required to define control panels, reader and IO points, setup access schedules and profiles. A Maestro server on it's own has a minimal user interface and serves primarily as the communications engine for control panels and clients.

# Application Guidelines

There are several guidelines Sage application developers should adhere to in order for their applications to integrate smoothly with Maestro. These guidelines are presented below.

## 1. Working Directory

It is not safe to assume that the current working directory, or the directory from which the application is started is the directory in which Maestro and other Sage based applications are loading and storing data. The `SgGetWorkingDirectory` call is provided for determining this.

## 2. Data Storage

Maestro and other Sage applications store all related data files in the Data directory. This is a sub-directory of the working directory. Maestro provides a file backup and restore utility which will backup all files in the Data directory. For this reason, it is suggested that data files associated with your application be stored there as well.

## 3. SgOpenSession

Sage applications should avoid opening and closing sessions with the Maestro server repeatedly during execution. Opening a session with the server causes the server to re-synchronize all connected clients with the current reader and IO point states. As a system becomes larger, this adds unnecessary overhead to the server. It is preferable to open the session once, when the application starts, and to close it when it is terminating.

## 4. SgSetMsgMask

If your application is not processing real time transaction information by polling `SgGetMsg`, ensure that you have masked out event buffering using `SgSetMsgMask` or buffer overflow problems may result.

## Where to Begin

The best place to start learning about Sage is by reading through the tutorials in the next section. These tutorials present a subset of the API's and work towards creating working applications, which demonstrate some of the functionality of the API set. Not all API functions are covered in these tutorials so the reader is directed to the API reference for a more complete treatment of the subject matter.

# Sage Tutorial

# Tutorial Requirements

The tutorials that follow present Sage programming concepts and samples. These samples were developed using Microsoft Visual C++ Version 6.0. The reader is assumed to have working knowledge of CP30/40 functionality and either PassMaster or Maestro. In order to execute and run some of these programs, a CP30/40 panel with an IO board and card reader is required.

The sample Sage programs were all developed as WIN32 console applications to focus the examples on the API calls themselves and not obscure them in MFC code.

## Initializing Sage

## Sage Initialization

To begin using Sage, three Sage initialization routines must be called to setup and connect with a Maestro server. The first `SgInitialize` allows Sage to initialize its internal data structures and register the application name with the DLL. The passing of the application name allows Sage to identify itself to Maestro as well as create a log file in which it can record any runtime exceptions. The second function `SgOpenSession` is called to create a connection with the Maestro server.

When connected with a Maestro server, a Sage application can receive events and issue commands to control and program field devices. The Sage DLL manages the connection between the application and the Maestro server.

The final function `SgLogin` logs the operator into Sage, and is required before subsequent calls can be made to the DLL.

## Establishing a Connection

The following code demonstrates Sage initialization;

```
#include <windows.h>
#include <Sage.h>

void main()
{
    // first initialize the DLL
    if (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting ....\n");
        exit(1);
    }

    // next startup a connection to the Maestro server
    if (SgOpenSession("SERVER") != SG_SUCCESS) {
        printf("Unable to open a session to the Maestro server, exiting...\n");
        exit(1);
    }

    // login using default user
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
        printf("Unable to login, exiting...\n");
        SgCloseSession();
        exit(1);
    }

    // do some application specific stuff here
    .....

    // terminate the connection to the server
    SgCloseSession();
}
```

In the second call `SgOpenSession`, the application passes the server name to the DLL. This name is the machine name as defined under the Network Neighborhood properties Identification tab. The name is used to identify the host computer which has created the named pipe. A successful return from `SgOpenSession` does not imply that a connection has been established, but merely that the DLL was successful in creating the thread which opens and manages the pipe between the application and the Maestro server.

The third call logs an operator into Sage, and audits the event at the Maestro server. `SgLogin` takes three arguments, the first two being the user name and password. Operator login information is set up through Maestro via the *Operator* command from the *Edit* menu. The last argument identifies the location of the login. This can be any name that serves to identify the location of the login (see `SgGetLocalName`).

The fourth and final call in this example is `SgCloseSession`. This function terminates the connection to the Maestro server and frees any resources allocated by the Sage DLL. All applications must call `SgCloseSession` to terminate their applications gracefully.

## Querying Available Devices

## Enumerating Defined Card Readers

Once an application has completed initialization, one of the first things it will probably want to do is determine what field devices are out there. The function `SgGetDefinedReaders` accomplishes part of this task by enumerating all defined card readers.

```
int SgGetDefinedReaders(SGDEVDEF *list, int bufsize)
```

The prototype above shows that this function takes two arguments, the first of type `SGDEVDEF`. This simple structure's definition is shown below.

```
typedef struct devdef {  
    int    Id;  
    int    Panel;  
    int    Trunk;  
    char   Name[NAME_LGTH + 1];  
} SGDEVDEF;
```

It consists of three integer variables, *Id* which represents the logical card reader number assigned to the device, *Panel*, the devices control panel's address, *Trunk*, the devices control panel trunk number, and a character array *Name* which holds the card reader name. The second argument *bufsize* indicates the number of elements present in the first argument. This permits Sage to determine if the passed buffer is large enough to hold the defined card reader list. `SgGetDefinedReaders` returns the actual number of entries stored in the list.

The following example demonstrates the use of `SgGetDefinedReaders`;

```
#include <windows.h>  
#include <Sage.h>  
  
int          ReaderCount;  
SGDEVDEF    Readers[MAX_READERS];  
  
void main()  
{  
    // first initialize the DLL  
    if (SgInitialize("MyApp") != SG_SUCCESS) {  
        printf("Unable to initialize the DLL, exiting ....\n");  
        exit(1);  
    }  
  
    // next startup a connection to the Maestro server  
    if (SgOpenSession("SERVER") != SG_SUCCESS) {  
        printf("Unable to open a session to the Maestro server, exiting...\n");  
        exit(1);  
    }  
  
    // login using default user  
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {  
        printf("Unable to login, exiting...\n");  
        SgCloseSession();  
    }  
}
```

```
        exit(1);
    }

    // load up a list of the defined card readers
    ReaderCount = SgGetDefinedReaders(Readers, MAX_READERS);

    // terminate the connection to the server
    SgCloseSession();
}
```

## Enumerating Defined Input and Output Points

As noted in the previous section, applications will most likely need knowledge of defined input and output points. To this end, the function `SgGetDefinedInputs` and `SgGetDefinedOutputs` are provided. Their prototypes are listed below.

```
int SgGetDefinedInputs(SGDEVDEF *list, int bufsize)
```

```
int SgGetDefinedOutputs(SGDEVDEF *list, int bufsize)
```

Both functions take identical arguments and return defined device counts as explained in the last section. Our continuing sample application is listed below.

```
#include <windows.h>
#include <Sage.h>

int          ReaderCount;
SGDEVDEF    Readers[MAX_READERS];

int          InputCount, OutputCount;
SGDEVDEF    Inputs[MAX_POINTS], Outputs[MAX_POINTS];

void main()
{
    // first initialize the DLL
    If (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting ....\n");
        exit(1);
    }

    // next startup a connection to the Maestro server
    If (SgOpenSession("SERVER") != SG_SUCCESS) {
        printf("Unable to open a session to the Maestro server, exiting...\n");
        exit(1);
    }

    // login using default user
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
        printf("Unable to login, exiting...\n");
        SgCloseSession();
        exit(1);
    }

    // load up a list of the defined card readers
    ReaderCount = SgGetDefinedReaders(Readers, MAX_READERS);

    // load up a list of the defined input points
    InputCount = SgGetDefinedInputs(Inputs, MAX_POINTS);

    // load up a list of the defined output points
    OutputCount = SgGetDefinedOutputs(Outputs, MAX_POINTS);

    // terminate the connection to the server
```

```
SgCloseSession();  
}
```

## Receiving Maestro Events

## The Sage Event Queue

One of the more useful features of the Sage toolkit is the ability to receive real time events from the field. This is accomplished through an event queue, which Sage manages. An application can choose to queue all events, no events or certain subsets through the use of the `SgSetMsgMask` function. By default, Sage queues all events and it is the responsibility of the application to retrieve events from the queue during execution. The mechanism by which an application retrieves events is the function `SgGetMsg`. The prototype is listed below.

```
int SgGetMsg(SGEVMSG *ptr)
```

`SgGetMsg` takes a single argument of type `SGEVMSG`. The definition for this structure is listed below.

```
typedef struct evmsgtag {  
    USHORT    MsgCode;  
    USHORT    Id;  
    UCHAR     U      Status;  
    USHORT    AStatus;  
    USHORT    BStatus;  
    UCHAR     IOStatus;  
    USHORT    OpData1;  
    USHORT    OpData2;  
    ULONG     OpData3;  
    WHEN     When;  
    PERDATA   Card;  
} SGEVMSG;
```

Each element of the structure is described below.

### **MsgCode**

This 16 bit integer value contains a code which uniquely identifies the event. These codes are described in detail in the API reference section.

### **Id**

This 16 bit integer value identifies the device where the event originated. This may signify a card reader id or an input or output id. The device type can be determined from the type of message.

### **Ustatus**

This eight value contains a status bit mask of the CP30/40 control panel status. The following bits may be set or reset.

<b>OLD_XACTION</b>	The event is over 2 minutes old.
<b>SYS_NOT_INIT</b>	The control panel is not initialized.
<b>BOX_TAMPER</b>	The panel box tamper switch is open.
<b>RDRA_DC_STATUS</b>	The A reader door contact is open.

**RDRB\_DC\_STATUS** The B reader door contact is open.

### Astatus and BStatus

These two 16 bit values contain a status mask for the panel A and B readers respectively. The following bits may be set or reset.

<b>RDR_GLOBAL_APB</b>	The reader is operating in Global Anti-Passback mode.
<b>RDR_FORCED</b>	There is a forced entry alarm at the reader.
<b>RDR_HELD_OPEN</b>	There is a door held open alarm at the reader.
<b>RDR_TAMPER</b>	These is a tamper alarm at the reader.
<b>RDR_UNLOCK</b>	The reader is currently unlocked.
<b>RDR_SYSCODE</b>	The reader is operating in system code mode.
<b>RDR_DUAL_CUST</b>	The reader is operating in dual custody mode.
<b>RDR_LOCKOUT</b>	The reader is operating in lock out mode.
<b>RDR_OFFLINE</b>	The reader is currently in communications failure.

### IOStatus

This eight bit value contains a status mask for an IO point. The following bits may be set or reset depending on the point type. For input points, the following values apply.

<b>PT_SECURE</b>	The input point is secure.
<b>PT_ALARM</b>	The input point is in alarm.
<b>PT_TROUBLE</b>	The input point is in a trouble open/closed state.
<b>PT_OFFLINE</b>	The input point is in communications failure.
<b>PT_SHUNTED</b>	The input is shunted.

For output points, the following states apply.

<b>PT_OFF</b>	The output is off or deactivated.
<b>PT_ON</b>	The output point is on or activated.

### Opdata1, Opdata2, and Opdata3

These optional data fields contain information that is dependent on the type of event received. Opdata1, and Opdata2 are unsigned short fields which could contain schedule or download block numbers. Opdata3 is an unsigned long field which may contain a 5 digit system code, for example.

### When

This element is a structure of type *WHEN* indicating when the event occurred. The definition of *WHEN* is presented below.

```
typedef struct whentag {
    unsigned short    hour;
    unsigned short    minute;
    unsigned short    day;
    unsigned short    month;
    unsigned short    year;
```

} WHEN;

### Card

This element is a structure of type *PERDATA* which contains associated card holder data, if present. The definition of *PERDATA* is presented below along with a description of each field.

```
typedef struct perdatatag {
    int          card_id1;
    int          card_id2;
    char         name1[NAME_LGTH + 1];
    char         name2[NAME_LGTH + 1];
    char         user1[USER_LGTH + 1];
    char         user2[USER_LGTH + 1];
    char         user3[USER_LGTH + 1];
    char         user4[USER_LGTH + 1];
    char         user5[USER_LGTH + 1];
    char         user6[USER_LGTH + 1];
    char         user7[USER_LGTH + 1];
    char         user8[USER_LGTH + 1];
    char         user9[USER_LGTH + 1];
    char         user10[USER_LGTH + 1];
    char         status1;
    char         status2;
    short int    extend_operation;
    short int    unlock_time;
    short int    dho_time;
} PERDATA;
```

#### **card\_id1** and **card\_id2**

If the event is associated with the use of a card or cards, the card number(s) are present in these integer elements. Examples of events with single card numbers are access granted or card validate events. Double card number events are restricted to dual custody operation.

#### **name1** and **name2**

These two character arrays hold the names associated with the card holders of *card\_id1* and *card\_id2*.

#### **user1** through **user10**

These 10 fields consist of the user definable field data associated with *card\_id1*.

#### **status1** and **status2**

These two character fields hold the card status associated with the card holders of *card\_id1* and *card\_id2*.

#### **extend\_operation**, **unlock\_time**, and **dho\_time**

These handicap control fields contain the extended unlock and door held open times associated with **card\_id1**, if the card holder is identified as handicapped via the *extend\_operation* flag.

## Event Types

A complete reference of all Maestro event codes is presented in the API Reference section. For the purposes of this tutorial, we will consider three events and develop a sample application around them. The first is the door held open event, which results when a valid access granted occurs, followed by a door contact open without a subsequent closure within a pre-programmed time.

### **DOOR\_HELD\_OPEN**

The door has been held open and has gone into alarm.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card**

Listed above are the relevant fields in the SGEVMSG structure. Because this is a card reader related event, the *IoStatus* member is not applicable. The DOOR\_HELD\_OPEN event has no associated optional data in Opdata1, Opdata2, or Opdata3. Also, there is no card-related data associated with this event so *Card* is not used as well.

The second event is the forced entry message. This event occurs, when the door contact of a locked door is opened in the absence of valid access granted or request to exit.

### **FORCED\_ENTER**

The door contact has opened causing a forced entry alarm.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card**

Like the door held open event, io point, optional, and card holder information is not present.

The third event we will consider is an input point state change. These occur when an input point changes state from one of secure, alarm or trouble to one of secure, alarm or trouble.

### **INPUT\_PT\_STATE\_CHANGE**

An input point has changed state. *IoStatus* will contain the new state of the input point and will be one of the following, PT\_SECURE, PT\_ALARM, PT\_TROUBLE.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card**

Notice that in this event message, the *IoStatus* member is valid and contains the new state of the input point.

## A Sample Alarm Monitor Application

In the following example we bring together the concepts discussed above into a simple alarm monitoring application. This program monitors door held open, forced entry and input point alarms and reports them and their location on screen.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <Sage.h>

int          ReaderCount;
SGDEVDEF    Readers[MAX_READERS];
int          InputCount, OutputCount;
SGDEVDEF    Inputs[MAX_POINTS], Outputs[MAX_POINTS];

char *GetDevName(int type, int id)
{
    int      i;

    switch(type) {
        case 0:
            for(i = 0; i < ReaderCount; i++) {
                if (Readers[i].Id == id)
                    return(Readers[i].Name);
            }
            return("");

        case 1:
            for(i = 0; i < InputCount; i++) {
                if (Inputs[i].Id == id)
                    return(Inputs[i].Name);
            }
            return("");

        case 2:
            for(i = 0; i < OutputCount; i++) {
                if (Outputs[i].Id == id)
                    return(Outputs[i].Name);
            }
            return("");
    }
    return("");
}

void main(int argc, char **argv)
{
    SGEVMSG      Event;

    // first initialize the DLL
    if (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting ....\n");
        exit(1);
    }
}
```

```

}

// next startup a connection to the Maestro server
if (SgOpenSession("SERVER") != SG_SUCCESS) {
    printf("Unable to open a session to the Maestro server, exiting...\n");
    exit(1);
}

// login using default user
if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
    printf("Unable to login, exiting...\n");
    SgCloseSession();
    exit(1);
}

// load up a list of the defined card readers
ReaderCount = SgGetDefinedReaders(Readers, MAX_READERS);

// load up a list of the defined input points
InputCount = SgGetDefinedInputs(Inputs, MAX_POINTS);

// load up a list of the defined output points
OutputCount = SgGetDefinedOutputs(Outputs, MAX_POINTS);

// next enter a loop monitoring Maestro events
while(TRUE) {
    // wait for a message or a terminating keystroke
    while(!_kbhit() && SgGetMsg(&Event) == SG_NONE_AVAILABLE)
        Sleep(500L);

    // if keyboard hit, break from loop
    if (_kbhit())
        break;

    // check for message codes code considered as alarms
    switch(Event.MsgCode) {
        case DOOR_HELD_OPEN:
            printf("\007Door Held Open alarm at reader %s\n",
                GetDevName(0, Event.Id));
            break;

        case FORCED_ENTER:
            printf("\007Forced Entry alarm at reader %s\n",
                GetDevName(0, Event.Id));
            break;

        case INPUT_PT_STATE_CHANGE:
            if (Event.IOStatus == PT_ALARM)
                printf("\007Input Point alarm at input %s\n",
                    GetDevName(1, Event.Id));
            break;
    }
}

// terminate the connection to the server

```

```
printf("Terminating...\n");  
SgCloseSession();  
}
```

## Querying Connection and Device Status

## Client Connection Status

After successful initialization of Sage through `SgInitialize` and `SgOpenSession`, an application may wish to determine the current status of its connection with a Maestro server. As mentioned previously, Sage manages the connection between the server and itself, and the server may not always be available as a Sage application is executing. This can easily be determined by calling the `SgGetConnectionStatus` function.

**int SgGetConnectionStatus(void)**

This function returns one of the following return codes.

<b>SG_NO_CONNECTION</b>	if there is no connection with the Maestro server.
<b>SG_INVALID_KEY</b>	if the run time key of the Sage client does not match the key of the Maestro server's.
<b>SG_NO_LOGIN</b>	if an operator has not logged in to Sage
<b>SG_SUCCESS</b>	if the Sage client is connected with the Maestro Server.

## Device Status

Although it is possible to track the state of card reader and io devices by monitoring the event queue, Sage already does this for you. Two commands are available for retrieving the status of a field device;

**USHORT SgGetReaderStatus(int Id, USHORT \*Status)**

**USHORT SgGetIOPointStatus(int Id, USHORT \*Status)**

The first function SgGetReaderStatus takes a reader id number as it's first argument and a pointer to an unsigned short integer as it's second. If successful, the variable *Status* is updated with a bit mask indicating the current status of the card reader. See the description for *AStatus* and *BStatus* under the section The Sage Event Queue, for a description of the bit mask values.

The second function SgGetIOPointStatus returns status information for input and output devices. Like the first function, SgGetIOPointStatus takes an IO point id number as it's first argument and a pointer to an unsigned short integer as it's second. If successful, the variable *Status* is updated with a constant indicating the current state of the IO point. See the description for *IOStatus* under the section The Sage Event Queue, for a description of the constant values.

## Querying Other Objects

## SGLISTDEF Structure

Several routines are available to enumerate specific types of data under Maestro. All of these routines (with the exception of the device enumeration functions) employ the data structure SGLISTDEF. The generic data structure serves merely to provide a means of listing the names of defined data and optionally an integer value dependant on the data in question. The structure definition is presented below.

```
// list definition record
typedef struct listdef {
    int    Id;                // Id number if applicable
    int    Optional;         // optional integer value
    char   Name[NAME_LGTH + 1]; // record or data name
} SGLISTDEF;
```

# Controlling Card Readers

## Card Reader Commands

One activity many Sage applications will engage in is control of card readers. To support this, Sage provides seven functions for locking, unlocking and locking out card readers. The first function is the unlock command whose prototype is displayed below.

**int SgUnlockReader(USHORT \*, int)**

This function takes as its first argument an array of short integers. Each integer element refers to a defined card reader identification number as returned from SgGetDefinedReaders. The second argument indicates the number of elements in the list with valid data. SgUnlockReader, validates each id number before processing the command itself. SgUnlockReader returns one of the following return codes;

<b>SG_BAD_PARAMETER</b>	if one of the id numbers is undefined.
<b>SG_NO_CONNECTION</b>	if there is no connection with the Maestro server.
<b>SG_FAILURE</b>	if memory could not be allocated to submit the command
<b>SG_NO_LOGIN</b>	if an operator has not logged in to Sage
<b>SG_SUCCESS</b>	if the command was successfully submitted to the server.

Note that *SG\_SUCCESS* only indicates successful command submission, not successful command execution.

The other reader lock commands are;

**int SgUnlockMReader(USHORT \*list, int count)**

**int SgRelockReader(USHORT \*list, int count)**

The parameter lists and return codes are identical to SgUnlockReader. The first function, SgUnlockMReader unlocks a reader for a pre-programmed interval of time. The second function SgRelockReader re-locks a reader.

The reader lock-out commands are presented below.

**int SgLockoutReader(USHORT \*list, int count)**

**int SgRemoveLockoutReader(USHORT \*list, int count)**

Like the reader lock commands, the lock-out commands take identical parameter lists and return identical return codes. The first function, SgLockoutReader sets the card reader into a mode where all cards are denied access regardless of the normal access privilege. The second command SgRemoveLockoutReader removes the lock out mode of a card reader permitting normal card access once again.

There are two additional reader related commands, which operate only on controllers equipped with elevator control boards. These are the floor lock and re-lock commands.

**int SgLockElevatorReader(int id, int floor)**

**int SgUnlockElevatorReader(int id, int floor)**

Both of these commands take a single elevator reader id number as the first parameter, and a floor relay number as the second. Elevator controllers may be configured as either 16, 32, 48, or 64 floor units, and consequently, the allowable floor relay parameters will differ.

The first function SgLockElevatorReader will re-lock the floor specified by the floor parameter, at the reader specified by id. The second function SgUnlockElevatorReader will lock the specified floor.

## A Sample Reader Command Application

In the following example, we use the reader lock commands and a simple menu to allow a user to manually control the card reader lock state.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <Sage.h>

int          ReaderCount;
SGDEVDEF    Readers[MAX_READERS];

BOOL IsValid(int id)
{
    int      i;

    for(i = 0; i < ReaderCount; i++) {
        if (Readers[i].Id == id)
            return(TRUE);
    }
    return(FALSE);
}

void main(int argc, char **argv)
{
    BOOL      done;
    USHORT   id;
    SGEVMSG   Event;

    // first initialize the DLL
    if (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting ....\n");
        exit(1);
    }

    // next startup a connection to the Maestro server
    if (SgOpenSession("SERVER") != SG_SUCCESS) {
        printf("Unable to open a session to the Maestro server, exiting...\n");
        exit(1);
    }

    // do not queue events
    SgSetMsgMask(SG_TYPE_NONE);

    // login using default user
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
        printf("Unable to login, exiting...\n");
        SgCloseSession();
        exit(1);
    }

    // load up a list of the defined card readers
    ReaderCount = SgGetDefinedReaders(Readers, MAX_READERS);
}
```

```

// next display a menu and process key strokes
done = FALSE;
while(!done) {
    printf("\n\nEnter 'U' to unlock a door\n");
    printf("      'M' to unlock momentarily\n");
    printf("      'R' to relock a door\n");
    printf("      'Q' to Quit\n\n");

    // wait for a key press
    switch(_getch()) {
        case 'u':
        case 'U':
            printf("Enter reader number (1 to 9): ");
            id = _getche() - '0';
            if (!IsValid(0, id)) {
                printf("\n\007Invalid reader number\n");
                break;
            }
            if (SgUnlockReader(&id, 1) != SG_SUCCESS)
                printf("\n\007Unlock command failed\n");
            break;

        case 'r':
        case 'R':
            printf("Enter reader number (1 to 9): ");
            id = _getche() - '0';
            if (!IsValid(0, id)) {
                printf("\n\007Invalid reader number\n");
                break;
            }
            if (SgRelockReader(&id, 1) != SG_SUCCESS)
                printf("\n\007Unlock command failed\n");
            break;

        case 'm':
        case 'M':
            printf("Enter reader number (1 to 9): ");
            id = _getche() - '0';
            if (!IsValid(0, id)) {
                printf("\n\007Invalid reader number\n");
                break;
            }
            if (SgUnlockMReader(&id, 1) != SG_SUCCESS)
                printf("\n\007Unlock command failed\n");
            break;

        case 'q':
        case 'Q':
            done = TRUE;
            break;

        default:
            printf("Wrong key pressed\n");
            break;
    }
}

```

```
}  
  
// terminate the connection to the server  
printf("Terminating...\n");  
SgCloseSession();  
}
```

# Controlling Inputs and Outputs

## Input Point Commands

The CP30/40 control panels do not support the notion of input point shunting. It is the responsibility of an application to implement this feature locally. Essentially, shunting means nothing more than ignoring input point state changes. While Maestro clients implement shunting internally through the *local shunt* commands, the Maestro server supports the notion of a global network shunt. This basically means that the Maestro server no longer reports input point state changes to clients while a global network shunt is in effect.

Sage supports Maestro global shunting through three commands;

**int SgShuntInput(USHORT \*list, int count)**

**int SgShuntMInput(USHORT \*list, int count, int duration)**

**int SgUnshuntInput(USHORT \*list, int count)**

Like the card reader commands discussed previously, the global shunt commands take two arguments (three for the momentary version), the first being a device id list. This list must consist of valid input point id numbers or else the function call will be rejected. The second parameter specifies the number of valid elements in the list. For the momentary command, the duration in seconds is specified as the third argument.

The first function SgShuntInput will shunt one or more inputs until further notice. The second function SgShuntMInput shunts one or more input points for a variable duration. The second function SgUnshuntInput removes a global shunt from an input.

Each of these functions returns one of the following return codes;

<b>SG_BAD_PARAMETER</b>	if one of the id numbers is undefined.
<b>SG_NO_CONNECTION</b>	if there is no connection with the Maestro server.
<b>SG_FAILURE</b>	if memory could not be allocated to submit the command.
<b>SG_NO_LOGIN</b>	if an operator has not logged in to Sage
<b>SG_SUCCESS</b>	if the command was successfully submitted to the server.

## Output Point Commands

Sage supports output point activation and de-activation through three function calls;

**int SgActivateOutput(USHORT \*list, int count)**

**int SgActivateMOutput(USHORT \*list, int count, int duration)**

**int SgDeactivateOutput(USHORT \*list, int count)**

Like the input shunt and card reader commands discussed previously, the output point functions take two arguments (three for the momentary version), the first being a device id list. This list must consist of valid output point id numbers or else the function call will be rejected. The second parameter specifies the number of valid elements in the list. For the momentary command, the duration in seconds is specified as the third argument.

The first function SgActivateOutput will activate one or more outputs until further notice. The second function SgActivateMOutput activates one or more output points for a variable duration. The second function SgDeactivateOutput de-activates an output point.

Each of these functions returns one of the following return codes;

<b>SG_BAD_PARAMETER</b>	if one of the id numbers is undefined.
<b>SG_NO_CONNECTION</b>	if there is no connection with the Maestro server.
<b>SG_FAILURE</b>	if memory could not allocated to submit the command.
<b>SG_NO_LOGIN</b>	if an operator has not logged in to Sage
<b>SG_SUCCESS</b>	if the command was successfully submitted to the server.

## A Sample IO Command Application

The following sample application demonstrates the input and output functions discussed previously. The example presents the user with a simple text mode menu and allows them to issue shunt and activation commands.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <Sage.h>

int          InputCount, OutputCount;
SGDEVDEF    Inputs[MAX_POINTS], Outputs[MAX_POINTS];

BOOL IsValid(int type, int id)
{
    int      i;

    switch(type) {
        case 1:
            for(i = 0; i < InputCount; i++) {
                if (Inputs[i].Id == id)
                    return(TRUE);
            }
            return(FALSE);

        case 2:
            for(i = 0; i < OutputCount; i++) {
                if (Outputs[i].Id == id)
                    return(TRUE);
            }
            return(FALSE);
    }
    return(FALSE);
}

void main(int argc, char **argv)
{
    BOOL      done;
    USHORT    id;

    // first initialize the DLL
    if (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting ....\n");
        exit(1);
    }

    // next startup a connection to the Maestro server
    if (SgOpenSession("SERVER") != SG_SUCCESS) {
        printf("Unable to open a session to the Maestro server, exiting...\n");
        exit(1);
    }
}
```

```

// do not queue events
SgSetMsgMask(SG_TYPE_NONE);

// login using default user
if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
    printf("Unable to login, exiting...\n");
    SgCloseSession();
    exit(1);
}

// load up a list of the defined input points
InputCount = SgGetDefinedInputs(Inputs, MAX_POINTS);

// load up a list of the defined output points
OutputCount = SgGetDefinedOutputs(Outputs, MAX_POINTS);

// next display a menu and process key strokes
done = FALSE;
while(!done) {
    printf("\n\nEnter 'S' to shunt an input\n");
    printf("      'M' to shunt an input momentarily\n");
    printf("      'R' to remove a shunt\n");
    printf("      'A' to activate an output\n");
    printf("      'B' to activate an output momentarily\n");
    printf("      'D' to deactivate an output\n");
    printf("      'Q' to Quit\n\n");

    // wait for a key press
    switch(_getch()) {
        case 's':
        case 'S':
            printf("Enter input number (1 to 9): ");
            id = _getche() - '0';
            if (!IsValid(1, id)) {
                printf("\n\007Invalid input number\n");
                break;
            }
            if (SgShuntInput(&id, 1) != SG_SUCCESS)
                printf("\n\007Shunt command failed\n");
            break;

        case 'r':
        case 'R':
            printf("Enter input number (1 to 9): ");
            id = _getche() - '0';
            if (!IsValid(1, id)) {
                printf("\n\007Invalid input number\n");
                break;
            }
            if (SgUnshuntInput(&id, 1) != SG_SUCCESS)
                printf("\n\007Shunt command failed\n");
            break;

        case 'm':
        case 'M':
            printf("Enter input number (1 to 9): ");

```

```

        id = _getche() - '0';
        if (!IsValid(1, id)) {
            printf("\n\007Invalid input number\n");
            break;
        }
        if (SgShuntMInput(&id, 1, 10) != SG_SUCCESS)
            printf("\n\007Shunt command failed\n");
        break;

case 'a':
case 'A':
    printf("Enter output number (1 to 9): ");
    id = _getche() - '0';
    if (!IsValid(2, id)) {
        printf("\n\007Invalid output number\n");
        break;
    }
    if (SgActivateOutput(&id, 1) != SG_SUCCESS)
        printf("\n\007Output command failed\n");
    break;

case 'd':
case 'D':
    printf("Enter output number (1 to 9): ");
    id = _getche() - '0';
    if (!IsValid(2, id)) {
        printf("\n\007Invalid output number\n");
        break;
    }
    if (SgDeactivateOutput(&id, 1) != SG_SUCCESS)
        printf("\n\007Output command failed\n");
    break;

case 'b':
case 'B':
    printf("Enter output number (1 to 9): ");
    id = _getche() - '0';
    if (!IsValid(2, id)) {
        printf("\n\007Invalid output number\n");
        break;
    }
    if (SgActivateMOutput(&id, 1, 10) != SG_SUCCESS)
        printf("\n\007Output command failed\n");
    break;

case 'q':
case 'Q':
    done = TRUE;
    break;

default:
    printf("Wrong key pressed\n");
    break;
}
}

```

```
    // terminate the connection to the server
    printf("Terminating...\n");
    SgCloseSession();
}
```

# Working with Groups

## Card and Device Group Functions

Maestro implements the notion of groups, which are nothing more than a set of device or card identification numbers and a user defined title string. There are four different types of groups supported, reader, input, output and card groups. The definition for a group is presented below.

```
#define      MAX_ITEMS_PER_GROUP    200
typedef struct grouptag {
    char    name[NAME_LGTH + 1];
    int     members[MAX_ITEMS_PER_GROUP];
} SGGROUP;
```

Sage provides a set of four functions for manipulating these groups. Each of these functions is displayed below.

```
int SgGroupLoad(int type, int which, SGGROUP *group)
```

```
int SgGroupSave(int type, int which, SGGROUP *group)
```

```
int SgGroupDelete(int type, int which)
```

```
int SgGroupAdd(int type, SGGROUP *group)
```

All of these functions reference the *type* parameter to determine the type of group they are working with. The group types are defined in the Sage.h include file and are one of **INPUT\_GROUP**, **OUTPUT\_GROUP**, **READER\_GROUP** or **CARD\_GROUP** definitions.

The *which* parameter used by the load, save and delete functions, specifies the group number to act upon. The third parameter used by the load, save and add functions, is a pointer to a buffer of type SGGROUP. This buffer contains the group data to add, save or store the data from a load function.

These functions return one of the following return codes.

**SG\_SUCCESS**, if function was successful.

**SG\_NO\_LOGIN** if an operator has not logged in to Sage

**SG\_FAILURE**, if the operation failed or Sage is not initialized.

**SG\_BAD\_PARAMETER**, if the *type* or *which* parameters is out of range.

## A Sample Group Application

In the following example, we use the group load functions in conjunction with the reader lock-out commands and a simple menu to allow a user to manually control the card reader lock-out state.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <Sage.h>

void main(int argc, char **argv)
{
    BOOL        done;
    int         groupid;
    USHORT     list[MAX_ITEMS_PER_GROUP];
    SGGROUP    group;

    // first initialize the DLL
    if (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting....\n");
        exit(1);
    }

    // next startup a connection to the Maestro server
    if (SgOpenSession("SERVER") != SG_SUCCESS) {
        printf("Unable to open a session to the Maestro server, exiting...\n");
        exit(1);
    }

    // do not queue events
    SgSetMsgMask(SG_TYPE_NONE);

    // login using default user
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
        printf("Unable to login, exiting...\n");
        SgCloseSession();
        exit(1);
    }

    // next display a menu and process key strokes
    done = FALSE;
    while(!done) {
        printf("\n\nEnter 'L' to lockout a reader group\n");
        printf("      'R' to remove lockout from the group\n");
        printf("      'Q' to Quit\n\n");

        // wait for a key press
        switch(_getch()) {
            case 'l':
            case 'L':
                // prompt for a group and load it
                printf("Enter reader group number (1 to 9): ");
                groupid = _getche() - '0';
```

```

        if (SgGroupLoad(READER_GROUP, groupid, &group) !=
SG_SUCCESS) {
            printf("\n\007Unable to load reader group\n");
            break;
        }

        // convert it over to USHORT
        for(i = 0; i < MAX_ITEMS_PER_GROUP && group.members[i];
i++)
            list[i] = group.members[i];

        // lock out the readers
        if (SgLockoutReader(list, i) != SG_SUCCESS)
            printf("\n\007Lock out command failed\n");
        break;

    case 'r':
    case 'R':
        // prompt for a group and load it
        printf("Enter reader group number (1 to 9): ");
        groupid = _getche() - '0';
        if (SgGroupLoad(READER_GROUP, groupid, &group) !=
SG_SUCCESS) {
            printf("\n\007Unable to load reader group\n");
            break;
        }

        // convert it over to USHORT
        for(i = 0; i < MAX_ITEMS_PER_GROUP && group.members[i];
i++)
            list[i] = group.members[i];

        // lock out the readers
        if (SgRemoveLockoutReader(list, i) != SG_SUCCESS)
            printf("\n\007Ulock out remove command failed\n");
        break;

    case 'q':
    case 'Q':
        done = TRUE;
        break;

    default:
        printf("Wrong key pressed\n");
        break;
    }
}

// terminate the connection to the server
printf("Terminating...\n");
SgCloseSession();
}

```

## **Controlling Card Reader Access**

## Access Profiles

One of the primary roles of a Maestro system is the programming and control of access at specific doors. This is accomplished by assigning each card holder an access profile. An access profile consists of a set of access schedules, one for each card reader in the system. Since many card holders have identical access requirements, a single access profile can be created to manage them as a group. The definition of an access profile is presented in the section Access Profile Structure.

Sage provides access profile support through eight API calls,

**SgGetAccessProfile(char \*name, PROFILE \*profile)**

**SgAddAccessProfile(PROFILE \*profile)**

**SgUpdateAccessProfile(PROFILE \*profile)**

**SgOpenProfileDatabase(void);**

**SgCloseProfileDatabase(int handle);**

**SgGetFirstAccessProfile(int handle, PROFILE \*profile)**

**SgGetNextAccessProfile(int handle, PROFILE \*profile)**

**SgGetAccessProfileCount(void)**

SgGetAccessProfile takes as its first argument, a pointer to a character string denoting the access profile to retrieve. The second parameter is a pointer to a buffer of type PROFILE, which will store the access profile data if successful. SgAddAccessProfile and SgUpdateAccessProfile each take a single pointer argument of type PROFILE, which points to the profile data to add or update.

The second set of functions is provided for enumerating available access profiles. The functions SgOpenProfileDatabase and SgCloseProfileDatabase open and close the profile database respectively. SgGetFirstAccessProfile returns the profile data associated with the first defined profile in the access profile database. Calls to SgGetNextAccessProfile return the next defined access profile following the one previously read. SgGetAccessProfileCount returns the number of defined profiles in the access profile database.

These functions return one of the following return codes.

**SG\_SUCCESS**, if function was successful.

**SG\_NO\_LOGIN** if an operator has not logged in to Sage

**SG\_FAILURE**, if the operation failed or Sage is not initialized.

**SG\_EXISTS**, if an add is attempted and the profile exists

**SG\_NOT\_FOUND**, if an update is attempted and the profile does not exist, or a get first or next operation is attempted and the record(s) do not exist.

**SG\_BAD\_HANDLE**, if the handle is not open or is invalid.

**SG\_BAD\_PARAMETER**, if one or more of the access schedules is not with in the range 0 to 15.

## A Sample Access Profile Application

This sample application prompts the user for an access profile name. If the corresponding profile exists, it is retrieved and the associated reader schedules are displayed.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <Sage.h>

int          ReaderCount;
SGDEVDEF    Readers[MAX_READERS];

void main(int argc, char **argv)
{
    int      i;
    char     name[NAME_LGTH + 1];
    PROFILE profile;

    // first initialize the DLL
    if (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting ....\n");
        exit(1);
    }

    // next startup a connection to the Maestro server
    if (SgOpenSession("SERVER") != SG_SUCCESS) {
        printf("Unable to open a session to the Maestro server, exiting...\n");
        exit(1);
    }

    // do not queue events
    SgSetMsgMask(SG_TYPE_NONE);

    // login using default user
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
        printf("Unable to login, exiting...\n");
        SgCloseSession();
        exit(1);
    }

    // load up a list of the defined card readers
    ReaderCount = SgGetDefinedReaders(Readers, MAX_READERS);

    // process key strokes
    while(TRUE) {
        printf("Enter a profile name or RETURN to quit: ");
        scanf("%s", name);
        if (!name[0])
            break;

        // read the record
        switch(SgGetAccessProfile(name, &profile)) {
            case SG_SUCCESS:
```

```

        printf("\nProfile Name: %s\n", profile.name);
        for(i = 0; i < ReaderCount; i++)
            printf("  %s: %02d\n", Readers[i].Name,
                profile.rdr_schedules[Readers[i].Id - 1]);
        break;

    case SG_NOT_FOUND:
        printf("\n\007Profile not found\n\n");
        break;

    case SG_FAILURE:
        printf("\n\007Error reading profile\n\n");
        break;
    }
}

// terminate the connection to the server
printf("Terminating...\n");
SgCloseSession();
}

```

# The Personnel Database

## Personnel Database Functions

A variety of functions are available for retrieving and updating personnel records. Most of these functions take as an argument a structure of type `CARD`. A description of this structure is presented in the API reference topic `Personnel Record Structure`.

Personnel database routines fall into three basic categories, enumeration, retrieval and updating. The enumeration functions are presented below.

### **int SgOpenCardDatabase(int indexfield)**

`SgOpenCardDatabase` opens the personnel database for subsequent read operations.

### **int SgCloseCardDatabase(int handle)**

`SgCloseCardDatabase` closes the handle passed to this function.

### **int SgGetFirstCardRecord(int handle, CARD \*record)**

`SgGetFirstCardRecord` retrieves the first personnel record in the database and stores the data in *record*.

### **int SgGetNextCardRecord(int handle, CARD \*record)**

`SgGetNextCardRecord` retrieves the next personnel record in the database and stores the data in *record*.

### **int SgFindCardRecord(int handle, char \*key, CARD \*record)**

**SgFindCardRecord** searches and retrieves a card record based on key field search text. If successful, the data is stored in *record*.

### **int SgGetCardRecordCount(void);**

Although not strictly a retrieval function, it is commonly used when scanning the database. `SgGetCardRecordCount` returns the number of records in the database or -1 if an error occurred.

All of these functions (with the exception of *SgGetCardRecordCount*) return one of the following Sage return codes.

**SG\_BAD\_PARAMETER**, The card number argument was out of range. Depending on the system, either 1 to 19500, 1 to 40000, or 1 to 65000.

**SG\_NOT\_FOUND**, The record does not exist.

**SG\_FAILURE**, An error occurred while accessing the database.

**SG\_BAD\_HANDLE**, The handle value passed was not open or out of range.

**SG\_NO\_LOGIN**, if an operator has not logged in to Sage

**SG\_SUCCESS**, The record was successfully retrieved.

The enumeration routines require that the personnel database be opened and then accessed sequentially through the get first and get next calls. A simple retrieval routine, which reads records based on the card number is available and shown below.

### **int SgGetCardRecord(int number, CARD \*record)**

SgGetCardRecord retrieves the personnel record whose card number field is set to *number* and stores the data in *record*.

This function returns one of the following codes,

**SG\_BAD\_PARAMETER**, The card number argument was out of range. Depending on the system, either 1 to 19500, or 1 to 40000.

**SG\_NOT\_FOUND**, The record does not exist.

**SG\_FAILURE**, An error occurred while accessing the database.

**SG\_NO\_LOGIN**, if an operator has not logged in to Sage

**SG\_SUCCESS**, The record was successfully retrieved.

The personnel record update routines enforce the card status/access profile field restrictions discussed previously. In addition various range checks are performed on the records before they are committed to the database. The functions and their return codes are discussed below.

**int SgInitCardRecord(int number, CARD \*record)**

The function SgInitCardRecord initializes the personnel record by setting the id field to the value passed in *number*, setting the validate date field to today's date, the void field to January 1<sup>st</sup>, 2050, the card status to *Inactive*, and the access profile to *No Access*. All other fields are set to null.

**int SgUpdateCardRecord(int number, CARD \*record)**

The function SgUpdateCardRecord, updates the record identified by *number* with the data stored in *record*. If a change to the access profile has taken place, the card will be re-validated at the appropriate card readers.

**int SgAddCardRecord(int number, CARD \*record)**

SgAddCardRecord adds a new record to the personnel database. If the card status is ACTIVE, the card is validated at the appropriate card readers,

**int SgVoidCardRecord(int number)**

The function SgVoidCardRecord, voids the personnel record identified by *number* by setting the access profile to *No Access* and card status to INACTIVE and voiding the it at all card readers.

**int SgDeleteCardRecord(int number)**

SgDeleteCardRecord deletes the personnel record identified by *number* and voids it at all card readers.

Possible return codes from these functions are,

**SG\_BAD\_PARAMETER**, if the card number is out of range.

**SG\_PER\_VALIDATE\_DATE**, if validate date field range check failed.

**SG\_PER\_VOID\_DATE**, if void date field range check failed.

**SG\_PER\_STATUS**, if the card status field is invalid.

**SG\_PER\_PROFILE**, if the profile/status field combination is incompatible.

**SG\_BAD\_RECORD**, if an unknown return from the record validation code is encountered.

**SG\_NOT\_FOUND**, if the record does not exist.

**SG\_FAILURE**, if a database error occurred.

**SG\_NO\_CONNECTION**, if the server was not available to validate the card.

**SG\_NO\_LOGIN**, if an operator has not logged in to Sage

**SG\_SUCCESS**, if the record was updated and validated.

## A Sample Personnel Database Application

This sample application prompts the user for a card number. If the corresponding record exists, it is retrieved and the number, name and access profile fields are displayed. Notice that in this application, a connection to the Maestro server is not required and a call to `SgOpenSession` is not made.

```
#include <windows.h>
#include <stdio.h>
#include <conio.h>
#include <Sage.h>

void main(int argc, char **argv)
{
    int    cardnumber;
    CARD   record;

    // first initialize the DLL
    if (SgInitialize("MyApp") != SG_SUCCESS) {
        printf("Unable to initialize the DLL, exiting ....\n");
        exit(1);
    }

    // login using default user
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
        printf("Unable to login, exiting...\n");
        exit(1);
    }

    // process key strokes
    while(TRUE) {
        printf("Enter a card number or '0' to quit: ");
        scanf("%d", &cardnumber);
        if (!cardnumber)
            break;

        // read the record
        switch(SgGetCardRecord(cardnumber, &record)) {
            case SG_SUCCESS:
                printf("\nNumber      : %s\n", record.id);
                printf("Name        : %s\n", record.name);
                printf("Access Profile: %s\n\n", record.profile);
                break;

            case SG_NOT_FOUND:
                printf("\n\007Record not found\n\n");
                break;

            case SG_BAD_PARAMETER:
                printf("\n\007Card number out of range\n\n");
                break;

            case SG_FAILURE:
                printf("\n\007Error reading record\n\n");
                break;
        }
    }
}
```

}  
  }  
    }

## Retrieving Audit Trail Information

## Audit Trail Commands

Support for the Maestro audit trail file comes in the form of three functions for opening, reading and closing the audit trail. Although Maestro maintains audit trail information as a set of one or more files, these functions treat the file set as a single file and select information from it based on a date and time include range. The first function, `SgOpenAuditTrail` makes use of the *WHEN* data structure presented in the section The Sage Event Queue.

### **int SgOpenAuditTrail(WHEN \*Start, WHEN \*End)**

The function `SgOpenAuditTrail` takes a start and end date-time range and returns a handle to be used for the read and close functions. A return value of -1 indicates failure.

### **void SgCloseAuditTrail(int Handle)**

`SgCloseAuditTrail` closes the handle returned from `SgOpenAuditTrail`.

### **BOOL SgReadAuditTrail(int Handle, SGEVMSG \*Buffer)**

The function `SgReadAuditTrail` reads and returns the next record in the file set. *Handle* is a handle returned from `SgOpenAuditTrail`. *Buffer* is a structure of type *SGEVMSG* discussed in section The Sage Event Queue. If the function is successful, `SgReadAuditTrail` returns TRUE. Otherwise it returns FALSE.

## A Sample Audit Trail Application

This simple application opens and reads a set of records from the audit trail file. As noted previously, because a connection is not required to the Maestro server, only SgInitialize and SgLogin are called.

```
#include <windows.h>
#include <stdio.h>
#include <sage.h>

void main()
{
    int    handle;
    WHEN   From, To;
    SGEVMSG Record;

    // Initialize the DLL
    if (SgInitialize("AudView") != SG_SUCCESS) {
        printf("Sage initialization failed\n");
        exit(1);
    }

    // login using default user
    if (SgLogin("Cansec", "Cansec", "My Computer") != SG_SUCCESS) {
        printf("Unable to login, exiting...\n");
        SgCloseSession();
        exit(1);
    }

    // Initialize start and end date/time buffers
    From.hour = 0;
    From.minute = 0;
    From.day = 3;
    From.month = 12;
    From.year = 1997;
    To.hour = 0;
    To.minute = 0;
    To.day = 3;
    To.month = 12;
    To.year = 1997;

    // open the audit trail
    if ((handle = SgOpenAuditTrail(&From, &To)) < 0) {
        printf("SgOpenAuditTrail failed\n");
        exit(1);
    }

    // read the records
    while(SgReadAuditTrail(handle, &Record)) {
        printf("%02d:%02d %02d- %02d- %02d Code %03d Id %03d\n",
            Record.When.hour, Record.When.minute,
            Record.When.day, Record.When.month, Record.When.year,
            Record.MsgCode, Record.Id);
    }
    SgCloseAuditTrail(handle);
}
```

```
    exit(0);  
}
```

# The Sage API Reference

# Initialization Functions

# SgInitialize

## Prototype

```
int SgInitialize(const char *Appname)
```

## Overview

This function performs basic initialization of the DLL and registers the application name.

## Parameters

*AppName*, a character pointer to the application's name

## Returns

SG\_SUCESS, if initialization is successful

SG\_FAILURE, if valid configuration information could not found

## SgOpenSession

### Prototype

```
int SgOpenSession(const char *ServerName)
```

### Overview

Initializes the DLL and establishes a connection to a Maestro server.

### Parameters

*ServerName*, a pointer to a null terminated string denoting the Maestro server's machine name.

### Returns

SG\_SUCCESS, if a connection has been established.

SG\_FAILURE, if the SgInitialize has not been called or there is an error initializing the DLL.

## SgCloseSession

### **Prototype**

```
int SgCloseSession(void)
```

### **Overview**

SgCloseSession disconnects from the Maestro server and frees allocated resources.

### **Parameters**

None

### **Returns**

SG\_SUCCESS, currently always successful

# SgGetWorkingDirectory

## Prototype

```
void SgGetWorkingDirectory(char *path)
```

## Overview

SgGetWorkingDirectory, saves the current Maestro working directory into the buffer pointed to by *path*. This function permits applications to determine the actual location of the Maestro installation.

## Parameters

path, a pointer to a character buffer to hold the Maestro installation path.

## Returns

none

# SgGetRunTimeRecord

## Prototype

```
void SgGetRunTimeRecord(RTCFG *config)
```

## Overview

SgGetRunTimeRecord stores the current installation information for this machine's instance of Maestro. This function permits applications to determine the configuration of the Maestro installation.

## Parameters

config, a pointer to structure of type RTCFG. This structure is defined in Sage.h and consists of the following fields.

```
typedef struct rtcfgtag {  
    int          SystemCode; // system code  
    unsigned char MaxPanels; // max panels supported  
    unsigned char MaxStations; // max client connections supported  
    short int    Options;    // option flags (see Sage.h)  
} RTCFG;
```

## Returns

none

## SgGetDefinedReaders

### Prototype

```
int SgGetDefinedReaders(SGDEVDEF *list, int bufsize)
```

### Overview

SgGetDefinedReaders, loads a list of all defined readers, their identification numbers, control panel addresses and trunk lines into the *list* buffer. If the list is not large enough, only *bufsize* entries are added.

### Parameters

*list*, a pointer to an array of elements of type SGDEVDEF which will hold the list.

*bufsize*, the number of elements present in the list

### Returns

The actual number of entries stored in the list.

## SgGetDefinedInputs

### Prototype

```
int SgGetDefinedInputs(SGDEVDEF *list, int bufsize)
```

### Overview

SgGetDefinedInputs, loads a list of all defined inputs, their identification numbers, control panel addresses and trunk lines into the *list* buffer. If the list is not large enough, only *bufsize* entries are added.

### Parameters

*list*, a pointer to an array of elements of type SGDEVDEF which will hold the list.

*bufsize*, the number of elements present in the list

### Returns

The actual number of entries stored in the list.

## SgGetDefinedOutputs

### Prototype

```
int SgGetDefinedOutputs(SGDEVDEF *list, int bufsize)
```

### Overview

SgGetDefinedOutputs, loads a list of all defined outputs, their identification numbers, control panel addresses and trunk lines into the *list* buffer. If the list is not large enough, only *bufsize* entries are added.

### Parameters

*list*, a pointer to an array of elements of type SGDEVDEF which will hold the list.

*bufsize*, the number of elements present in the list

### Returns

The actual number of entries stored in the list.

# SgGetLocalName

## Prototype

```
void SgGetLocalName(char *name)
```

## Overview

SgGetLocalName stores the current Maestro workstation name. The network version of Maestro uses a client name to identify each instance of Maestro on the network. The name is also used to generate unique file names specific to that machine. This function permits applications to determine the current Maestro workstation name.

## Parameters

name, a pointer to a char array of NAME\_LGTH + 1 bytes. NAME\_LGTH is defined in Sage.h.

## Returns

The name if defined, otherwise the string is of zero length (empty).

# Events

## Message Structure and Event Types

## Message Structure

The Sage DLL manages a queue of events received from the Maestro server. An application retrieves events from the queue by calling the function `SgGetMsg`. `SgGetMsg` takes a single argument of type `SGEVMSG`. The definition for this structure is listed below.

```
typedef struct evmsgtag {
    USHORT    MsgCode;
    USHORT    Id;
    UCHAR     UStatus;
    USHORT    AStatus;
    USHORT    BStatus;
    UCHAR     IOStatus;
    USHORT    OpData1;
    USHORT    OpData2;
    ULONG     OpData3;
    WHEN     When;
    PERDATA   Card;
} SGEVMSG;
```

Each element of the structure is described below.

### MsgCode

This 16 bit integer value contains a code which uniquely identifies the event. These codes are described in detail in the next section.

### Id

This 16 bit integer value identifies the device where the event originated. This may signify a card reader id or an input or output id. The device type can be determined from the type of message.

### Ustatus

This eight bit value contains a status bit mask of the CP30/40 control panel status. The following bits may be set or reset.

<b>OLD_XACTION</b>	The event is over 2 minutes old.
<b>SYS_NOT_INIT</b>	The control panel is not initialized.
<b>BOX_TAMPER</b>	The panel box tamper switch is open.
<b>RDRA_DC_STATUS</b>	The A reader door contact is open.
<b>RDRB_DC_STATUS</b>	The B reader door contact is open.

### Astatus and BStatus

These two 16 bit values contain a status mask for the panel A and B readers respectively. The following bits may be set or reset.

<b>RDR_GLOBAL_APB</b>	The reader is operating in Global Anti-Passback mode.
<b>RDR_FORCED</b>	There is a forced entry alarm at the reader.
<b>RDR_HELD_OPEN</b>	There is a door held open alarm at the reader.

<b>RDR_TAMPER</b>	There is a tamper alarm at the reader.
<b>RDR_UNLOCK</b>	The reader is currently unlocked.
<b>RDR_SYSCODE</b>	The reader is operating in system code mode.
<b>RDR_DUAL_CUST</b>	The reader is operating in dual custody mode.
<b>RDR_LOCKOUT</b>	The reader is operating in lock out mode.
<b>RDR_OFFLINE</b>	The reader is currently in communications failure.

### IOStatus

This eight bit value contains a status mask for an IO point. The following bits may be set or reset depending on the point type. For input points, the following values apply.

<b>PT_SECURE</b>	The input point is secure.
<b>PT_ALARM</b>	The input point is in alarm.
<b>PT_TROUBLE</b>	The input point is in a trouble open/closed state.
<b>PT_OFFLINE</b>	The input point is in communications failure.
<b>PT_SHUNTED</b>	The input is shunted.

For output points, the following states apply.

<b>PT_OFF</b>	The output is off or deactivated.
<b>PT_ON</b>	The output point is on or activated.

### OpData1, OpData2, OpData3

Various types of optional data are associated with some event types. Examples are system codes, duration's, block numbers and schedule numbers. This additional information will be stored in one or more of these optional data fields.

### When

This element is a structure of type *WHEN* indicating when the event occurred. The definition of *WHEN* is presented below.

```
typedef struct whentag {
    unsigned short    hour;
    unsigned short    minute;
    unsigned short    day;
    unsigned short    month;
    unsigned short    year;
} WHEN;
```

### Card

This element is a structure of type *PERDATA* which contains associated card holder data, if present. The definition of *PERDATA* is presented below along with a description of each field.

```
typedef struct perdatatag {
    int                card_id1;
    int                card_id2;
    char               name1[NAME_LGTH + 1];
    char               name2[NAME_LGTH + 1];
}
```

```

char        user1[USER_LGTH + 1];
char        user2[USER_LGTH + 1];
char        user3[USER_LGTH + 1];
char        user4[USER_LGTH + 1];
char        user5[USER_LGTH + 1];
char        user6[USER_LGTH + 1];
char        user7[USER_LGTH + 1];
char        user8[USER_LGTH + 1];
char        user9[USER_LGTH + 1];
char        user10[USER_LGTH + 1];
char        status1;
char        status2;
short int   extend_operation;
short int   unlock_time;
short int   dho_time;
} PERDATA;

```

#### **card\_id1 and card\_id2**

If the event is associated with the use of a card or cards, the card number(s) are present in these integer elements. Examples of events with single card numbers are access granted or card validate events. Double card number events are restricted to dual custody operation.

#### **name1 and name2**

These two character arrays hold the names associated with the card holders of *card\_id1* and *card\_id2*.

#### **user1 through user10**

These 10 fields consist of the user definable field data associated with *card\_id1*.

#### **status1 and status2**

These two character fields hold the card status associated with the card holders of *card\_id1* and *card\_id2*.

#### **extend\_operation, unlock\_time, and dho\_time**

These handicap control fields contain the extended unlock and door held open times associated with **card\_id1**, if the card holder is identified as handicapped via the *extend\_operation* flag.

## Event Types

In the following sections we present a list of each of the event codes which may be present in the MsgCode element of a SGEVMSG structure. Each section attempts to categorize the events by type in order to ease searching in this help file. With each code definition, the fields containing valid data are displayed in bold. For the card holder data associated with the Card field (PERDATA), Card1 will refer to the fields Card.card\_id1, Card.name1, Card.status1, Card.user1 through Card.user10, Card.extend\_operation, Card.unlock\_time, and Card.dho\_time. Card2 will refer to the fields Card.card\_id2, Card.name2, and Card.status2.

For example;

## ACC\_DENIED\_SYSCODE

The access attempt was denied due to an invalid system code.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IOStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

<b>MsgCode</b>	<i>This field is valid</i>
<b>Id</b>	<i>This field is valid</i>
<b>Ustatus</b>	<i>This field is valid</i>
<b>Astatus</b>	<i>This field is valid</i>
<b>Bstatus</b>	<i>This field is valid</i>
<b>IOStatus</b>	<i>This field is not valid</i>
<b>OpData1</b>	<i>This field is not valid</i>
<b>OpData2</b>	<i>This field is not valid</i>
<b>OpData3</b>	<i>This field is valid</i>
<b>When</b>	<i>This field is valid</i>
<b>Card1</b>	<i>The following fields are valid,</i> <i>Card.card_id1, Card.name1, Card.user1 through Card.user10,</i> <i>Card.status1, Card.extend_operation, Card.unlock_time,</i> <i>Card.dho_time.</i>
<b>Card2</b>	<i>The following fields are not valid,</i> <i>Card.card_id2, Card.name2, Card.status2</i>

## Access Denied

### ACC\_DENIED\_SYSCODE

The access attempt was denied due to an invalid system code. OpData3 contains the system code.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_DENIED\_TZONE

The access attempt was denied because the card was used outside it's authorized time zone. OpData1 contains the access schedule number.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_DENIED\_PIN

The access attempt was denied due to an invalid PIN number entry.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_DENIED\_TIMED\_AP

The access attempt was denied due to a timed anti-passback violation.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_DENIED\_LOG\_AP

The access attempt was denied due to a logical anti-passback violation.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_DENIED\_RDR\_UNLK

The access attempt was denied while the card reader was unlocked. OpData1 contains the access schedule number.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_DENIED\_INTERLK**

The access attempt was denied due to a door interlock violation.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_DENIED\_TOO\_BIG**

The access attempt was denied because the card number greater than 19500 or 40000.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_DENIED\_GLOBAL\_APB**

The access attempt was denied due to a global anti-passback violation.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_DENIED\_LOCKOUT**

The access attempt was denied because the card reader is in lock-out mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_DENIED\_DURESS**

The access attempt was denied during a key pad duress code entry.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_DENIED\_SYSCODE**

The dual custody access attempt was denied because one of the cards system code was incorrect.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_DENIED\_TZONE**

The dual custody access attempt was denied because one of the cards was used outside it's time zone.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

**DC\_ACC\_DENIED\_PIN**

The dual custody access attempt was denied because one of the PIN numbers entered was incorrect.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

**DC\_ACC\_DENIED\_TIMED\_AP**

The dual custody access attempt was denied due to a timed anti-passback violation with one of the cards.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

**DC\_ACC\_DENIED\_LOG\_AP**

The dual custody access attempt was denied due to a logical anti-passback violation with one of the cards.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

**DC\_ACC\_DENIED\_RDR\_UNLK**

The dual custody access attempt was denied while the card reader was unlocked.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

**DC\_ACC\_DENIED\_INTERLK**

The dual custody access attempt was denied due to an interlock violation.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

**DC\_ACC\_DENIED\_TOO\_BIG**

The dual custody access attempt was denied due to presentation of a card with a number exceeding 19500 or 40000.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_DENIED\_SAME\_CARD**

The dual custody access attempt was denied because the same card was presented twice.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_DENIED\_DC\_ABORT**

The dual custody access attempt was denied because a second card was not presented.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **PATIENT\_WANDER**

A patient has exited through a monitored patient wander door loop.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **MULTI\_PATIENT\_WANDER**

One or more patients have exited through a monitored patient wander door loop. The door loop was unable to identify the patient tags that exited.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## Access Granted

### ACC\_GRANTED\_SYSCODE

A card was granted access while the reader was in system code mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_GRANTED\_CARD

A card was granted access.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_GRANTED\_CARD\_PIN

A card was granted access while used in conjunction with a valid PIN number.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_GRANTED\_RDR\_UNLK

A card was granted access while the card reader was unlocked.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_GRANTED\_TIMED\_AP

A card was granted access while the card reader was in timed anti-passback mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_GRANTED\_LOG\_AP

A card was granted access while the card reader was in logical anti-passback mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### ACC\_GRANTED\_INTERLK

A card was granted access while the card reader was in interlock mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_GRANTED\_INTRLK\_BY**

A card with unlock privilege removed interlock from this reader. Access was granted to a card while the card reader was in this by-passed mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_GRANTED\_GLOBAL\_APB**

A card was granted access while the card reader was in global anti-passback mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACC\_GRANTED\_DURESS**

A card was granted access at a keypad reader with a duress code entered.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_GRANTED\_SYSCODE**

Dual custody card access was granted while the reader was in system code mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_GRANTED\_CARD**

Dual custody card access was granted.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_GRANTED\_CARD\_PIN**

Dual custody card access was granted at a keypad reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_GRANTED\_RDR\_UNLK**

Dual custody card access was granted while the reader was unlocked.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_GRANTED\_TIMED\_AP**

Dual custody card access was granted while the reader was in timed anti-passback.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_GRANTED\_LOG\_AP**

Dual custody card access was granted while the reader was in logical anti-passback mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DC\_ACC\_GRANTED\_INTERLK**

Dual custody card access was granted while the reader was in interlock mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## Card Reader Events

### PREALERT\_SOUNDED

The card reader has sounded the son-alert warning of an impending door held open condition.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### DOOR\_HELD\_OPEN

The door has been held open and has gone into alarm.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### DOOR\_RECLOSED

The door contact has closed after a valid access attempt.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### FORCED\_ENTER

The door contact has opened causing a forced entry alarm.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### DOOR\_OPENED

The door contact has been opened following a valid access granted.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### EXIT\_BUTTON\_PRESSED

The request to exit button has been pressed, the alarm shunt enabled and the lock released (if unlock on RTE enabled) to allow access.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### READER\_UNLOCKED

The door has been unlocked.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **READER\_LOCKED**

The door has been re-locked.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **MOMENTARY\_UNLOCK**

The door has been unlocked for a pre-programmed interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **TAMPER\_SWITCH\_OPENED**

The card reader tamper switch has been opened.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **TAMPER\_SWITCH\_CLOSED**

The card reader tamper switch has been closed.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DOOR\_TIMES\_EXTENDED**

A handicap access granted occurred and the default unlock time was extended.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **CONTROL\_TAMPER\_OPENED**

The control panel box tamper switch was opened.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## **CONTROL\_TAMPER\_CLOSED**

The control panel box tamper switch was closed.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## IO Point Events

### INPUT\_PT\_STATE\_CHANGE

An input point has changed state. IOStatus will contain the new state of the input point and will be one of the following, PT\_SECURE, PT\_ALARM, PT\_TROUBLE.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### OUTPUT\_PT\_STATE\_CHANGE

An output point has changed state. IOStatus will contain the new output point state and will be set to either PT\_ON or PT\_OFF.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### OUTPUT\_PT\_TMP\_CHANGE

An output point has been activated for a pre-programmed interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### INPUT\_PT\_SHUNT\_CHANGE

An input point has been shunted. IOStatus will contain TRUE if the point has been shunted or FALSE if the shunt has been removed.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### INPUT\_PT\_TMP\_CHANGE

An input point has been shunted for a pre-programmed interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### INPUT\_LOCAL\_SHUNT

An input point has been shunted locally at a Maestro network client. IOStatus will contain TRUE if the point has been shunted or FALSE if the shunt has been removed.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **INPUT\_LOCAL\_TMP\_SHUNT**

An input point has been shunted locally at a Maestro network client for a pre-programmed interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## Programming Events

### SYS\_INIT

A control panel has been initialized with a system code. This may occur when a panel is brought on-line or in response to a panel download. OpData3 contains the system code.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### INTERLOCK\_BYPASSED

A card with unlock privilege has double swiped at the reader and bypassed interlock mode.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### INTERLOCK\_RESUMED

A card with unlock privilege has double swiped and resumed interlock at a bypassed reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### TIMED\_AP\_ON

Timed anti-passback mode has been enabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### TIMED\_AP\_OFF

Timed anti-passback mode has been disabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### LOCAL\_AP\_ON

Local anti-passback mode has been enabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **LOCAL\_AP\_OFF**

Local anti-passback mode has been disabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **GLOBAL\_AP\_ON**

Global anti-passback mode has been enabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **GLOBAL\_AP\_OFF**

Global anti-passback mode has been disabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **LOCKOUT\_ON**

Lock-out mode has been enabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **LOCKOUT\_OFF**

Lock-out mode has been disabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **SYSCODE\_MODE\_OFF**

System code mode has been disabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **SYSCODE\_MODE\_ON**

System code mode has been enabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **CARD\_VALIDATED**

A card has been validated at the card reader. OpData1 contains the access schedule number.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **UNLOCK\_SCHEDULE\_RCVD**

A new reader unlock schedule has been received by the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **HOLIDAY\_ACCESS\_SCHED**

The reader is now granting access based on the holiday entry in all access schedules.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **READER\_UNLOCK\_TIME**

A new unlock time was received by the card reader. OpData3 contains the new unlock interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **READER\_HELD\_OPEN\_TIME**

A new reader door held open time was received by the card reader. OpData3 contains the new held open interval. The value must be multiplied by 10, to determine the actual interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DUAL\_CUSTODY\_MODE**

Dual custody mode has been enabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## DUAL\_CUSTODY\_REMOVED

Dual custody mode has been disabled at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## REPORT\_EXIT\_BUTTON

Request to exit button reporting has been enabled at the reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## DONT\_REPORT\_EXIT\_BUTTON

Request to exit button reporting has been disabled at the reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## ENABLE\_PREALERT

Door held open pre-alert mode has been enabled at the reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## DISABLE\_PREALERT

Door held open pre-alert mode has been disabled at the reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## SET\_PREALERT\_TIME

A new door held open pre-alert time has been received by the card reader. OpData3 contains the new pre-alert interval. The value must be multiplied by 10, to determine the actual interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## REPORT\_ALL\_TRANSACTIONS

The card reader has been programmed to transmit access granted messages to the host.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **REPORT\_ALARMS\_ONLY**

The card reader has been programmed to not transmit access granted messages to the host.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **REPORT\_DOOR\_OPENCLOSE**

The card reader has been programmed to report door contact open/close messages to the host.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DONT\_REPORT\_OPENCLOSE**

The card reader has been programmed to not report door contact open/close messages to the host.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **INTERLOCK\_OFF**

Door interlock mode has been disabled.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **INTERLOCK\_ON**

Door interlock has been enabled.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ANTIPASS\_SET\_TO**

A new timed anti-passback duration has been programmed. OpData3 contains the new anti-passback interval.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **IN\_OUT\_ON**

Door In/Out mode has been enabled.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **IN\_OUT\_OFF**

Door In/Out mode has been disabled.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **HOLIDAY\_ENTRY**

Not used.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **UNLOCK\_SCHED\_CHANGED**

A new unlock schedule has been received at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **HOLIDAY\_SCHED\_CHANGED**

A new holiday schedule has been received at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **DATABASE\_DOWNLOAD**

A card database download command was received and executed by the reader/panel. OpData1 contains the block number. Block 0 contains the cards 1 through 100, block 1, the cards 101 through 200, etc.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **UNLK\_PRIV\_DOWNLOAD**

Card unlock privilege data was received by the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ELEV\_FLOOR\_UNLOCK**

One or more elevator floors have been unlocked.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ACCESS\_SCHED\_DNLD**

The card reader received a new access schedule.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **RDRA\_HOLIDAY\_UNLK\_SCHED**

The reader is now unlocking the door based on the holiday entry of the unlock schedule.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **RDRB\_HOLIDAY\_UNLK\_SCHED**

The reader is now unlocking the door based on the holiday entry of the unlock schedule.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **APB\_READER\_DEFINED**

The card reader has been programmed to participate in global anti-passback. OpData1 contains the reader's new global anti-passback level.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **UNLK\_RTE\_ON**

The card reader has been programmed to activate the lock output when the request to exit button is pressed.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **UNLK\_RTE\_OFF**

The card reader has been programmed not to activate the lock output when the request to exit button is pressed.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **BLOCK\_CLEARED**

A block of cards have been cleared from the control panel during a database download. OpData1 contains the starting block number, and OpData2 the ending block number. Block 0 contains the cards 1 through 100, block 1, the cards 101 through 200, etc.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **LOW\_BATTERY**

A low battery condition was detected in a Cotag prox card.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **EXT\_DBASE\_DOWNLOAD**

A card database download command was received and executed by the reader/panel. OpData1 contains the block number. Block 0 contains the cards 1 through 100, block 1, the cards 101 through 200, etc.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **EXT\_BLOCK\_CLEARED**

A block of cards have been cleared from the control panel during a database download. OpData1 contains the starting block number, and OpData2 the ending block number. Block 0 contains the cards 1 through 100, block 1, the cards 101 through 200, etc.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## KPAD\_MODE\_CHANGE

The card reader/keypad mode of operator has changed. OpData3 contains the new keypad mode encoded as follows. A value of 48 indicates that *Card Only* mode is enabled. A value of 49, *Card + PIN* mode, and 50, *PIN Only* mode.

MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2

## KPAD\_RETRY\_CHANGE

The keypad retry count has been changed. OpData3 contains the new retry count encoded as follows. New Count = OpData3 – 48.

**MsgCode, Id, Ustatus, Astatus, Bstatus,** IoStatus, OpData1, OpData2, **OpData3,** **When,** Card1, Card2

## KPAD\_ALG\_CHANGE

The keypad encryption algorithm has been changed. OpData3 contains the new algorithm encoded as follows. New Algorithm = OpData3 – 48 + 1.

**MsgCode, Id, Ustatus, Astatus, Bstatus,** IoStatus, OpData1, OpData2, **OpData3,** **When,** Card1, Card2

## KPAD\_RANGE\_CHANGE

The keypad card number bypass range has been changed.

**MsgCode, Id, Ustatus, Astatus, Bstatus,** IoStatus, OpData1, OpData2, OpData3, **When,** Card1, Card2

## ELEV\_PANEL\_DEFINED

The card reader/panel has been defined as an elevator reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus,** IoStatus, OpData1, OpData2, OpData3, **When,** Card1, Card2

## ELEV\_UNLK\_SCHED\_ASSIGN

Elevator floor unlock schedules have been assigned.

**MsgCode, Id, Ustatus, Astatus, Bstatus,** IoStatus, OpData1, OpData2, OpData3, **When,** Card1, Card2

#### **ELEV\_FLOOR\_GRP\_DNLD**

An elevator floor group has been downloaded to the reader. OpData1 + 1 contains the floor group number that was downloaded.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **ELEV\_SCHEDULED\_UNLK**

A group of one or more elevator floors have been unlocked.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **IO\_SCHED\_DNLD**

An output schedule has been received at the control panel. OpData1 contains the output schedule number that has been downloaded.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

#### **IO\_MAP\_DNLD**

A new io point definition table has been received by the control panel.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

## Miscellaneous Events

### COMM\_RESTORED

Communications have been restored at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### NO\_COMM

Communications have failed at the card reader.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### SYSTEM\_STARTUP

The Maestro server has come online.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card2**

### OPERATOR\_LOGON

An operator has logged on to a Maestro client. Note that only the Card.name1 field contains valid data.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card.name1, Card2**

### OPERATOR\_LOGOFF

An operator has logged off of a Maestro client. Note that only the Card.name1 field contains valid data.

**MsgCode, Id, Ustatus, Astatus, Bstatus, IoStatus, OpData1, OpData2, OpData3, When, Card1, Card.name1, Card2**

## Event Functions

## **SgGetMsg**

### **Prototype**

int SgGetMsg(SGEVMSG \*ptr)

### **Overview**

SgGetMsg retrieves a Maestro event message from the message queue.

### **Parameters**

*ptr*, a pointer to a buffer of type SGEVMSG to hold the retrieved message

### **Returns**

SG\_NONE\_AVAILABLE, if no messages are queued  
SG\_NO\_LOGIN, if an operator has not logged in to Sage  
SG\_SUCCESS, if a message was retrieved  
SG\_BAD\_PARAMETER, if the pointer is NULL

## SgSetMsgMask

### Prototype

```
void SgSetMsgMask(int mask)
```

### Overview

SgSetMsgMask sets an event mask, which determines the class or type of events that will be queued for the application. Note that SgOpenSession will reset the event mask, so ensure that calls to SgSetMsgMask occur only after a successful call to SgOpenSession.

### Parameters

A mask constructed using the boolean *or* operator. The following mask values are defined in Sage.h

SG_TYPE_NONE	No events mask.
SG_TYPE_GRANTED	Access granted type messages
SG_TYPE_DENIED	Access denied type messages
SG_TYPE_DOOR_ALARMS	Held open, forced entry, tamper
SG_TYPE_DOOR_MODES	Locked, unlocked, syscode, dual, interlock
SG_TYPE_PROGRAMMING	Validate, setup, download messages
SG_TYPE_SYSTEM	Start up, shut down messages
SG_TYPE_INPUT_STATE	Input state changes
SG_TYPE_OUTPUT_STATE	Output state changes
SG_TYPE_OPERATOR_EVENTS	Operator logon/off events
SG_TYPE_DC_EVENTS	Door Open/Re-closed and RTE messages
SG_TYPE_ALL	All event types

### Returns

None

## Status Functions

# SgGetConnectionStatus

## Prototype

int SgGetConnectionStatus(void)

## Overview

SgGetConnectionStatus queries the current connection status between Sage and the Maestro server.

## Parameters

None

## Returns

SG\_NO\_CONNECTION, if there is no connection with the Maestro server.

SG\_INVALID\_KEY, if the run time key of the Sage client does not match the key of the Maestro server's, or the Sage DLL version does not match the Maestro server's.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if the Sage client is connected with the Maestro Server.

# SgGetReaderStatus

## Prototype

int SgGetReaderStatus(int Id, USHORT \*Status)

## Overview

SgGetReaderStatus stores the current reader status in the second argument *Status*.

## Parameters

*Id*, a valid reader identification number

*Status*, an unsigned short integer which holds the reader status bit mask. Possible bit values for *Status* are;

<b>RDR_GLOBAL_APB</b>	The reader is operating in Global Anti-Passback mode.
<b>RDR_FORCED</b>	There is a forced entry alarm at the reader.
<b>RDR_HELD_OPEN</b>	There is a door held open alarm at the reader.
<b>RDR_TAMPER</b>	There is a tamper alarm at the reader.
<b>RDR_UNLOCK</b>	The reader is currently unlocked.
<b>RDR_SYSCODE</b>	The reader is operating in system code mode.
<b>RDR_DUAL_CUST</b>	The reader is operating in dual custody mode.
<b>RDR_LOCKOUT</b>	The reader is operating in lock out mode.
<b>RDR_OFFLINE</b>	The reader is currently in communications failure.

## Returns

SG\_SUCCESS, if *Id* references a valid reader id number, *Status* is updated.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_BAD\_PARAMETER, if *Id* does not reference a valid reader id number

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgGetReaderStatusList

## Prototype

```
int SgGetReaderStatusList(USHORT *pList, int Size)
```

## Overview

SgGetReaderStatusList stores the current status of all defined readers (or up to Size entries) in the array pointed to by pList. The order of status words in the array corresponds to the order of readers as returned by SgGetDefinedReaders.

## Parameters

*pList*, a pointer to an array of unsigned short integers which hold the reader status bit masks (see SgGetReaderStatus for bit mask definitions).

*Size*, an integer value specifying the number of elements in the array pointer to be *pList*.

## Returns

$\geq 0$ , number of elements stored in the list

-1, if an error occurred

# SgGetIOPointStatus

## Prototype

```
int SgGetIOPointStatus(int Id, USHORT *Status)
```

## Overview

SgGetIOPointStatus stores the current input or output point status in the second argument *Status*.

## Parameters

*Id*, a valid IO point identification number

*Status*, an unsigned short integer, which holds the IO point status. Possible input point values for *Status* are;

<b>PT_SECURE</b>	The input point is secure.
<b>PT_ALARM</b>	The input point is in alarm.
<b>PT_TROUBLE</b>	The input point is in a trouble open/closed state.
<b>PT_OFFLINE</b>	The input point is in communications failure.
<b>PT_SHUNTED</b>	The input is shunted.

Possible output point values for *Status* are;

<b>PT_OFF</b>	The output is off or deactivated.
<b>PT_ON</b>	The output point is on or activated.
<b>PT_OFFLINE</b>	The output point is in communications failure.

## Returns

SG\_SUCCESS, if *Id* references a valid input id number, *Status* is updated.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_BAD\_PARAMETER, if *Id* does not reference a valid input id number

# SgGetInputStatusList

## Prototype

```
int SgGetInputStatusList(USHORT *pList, int Size)
```

## Overview

SgGetInputStatusList stores the current status of all defined inputs (or up to Size entries) in the array pointed to by pList. The order of status words in the array corresponds to the order of inputs as returned by SgGetDefinedInputs.

## Parameters

*pList*, a pointer to an array of unsigned short integers which hold the input status (see SgGetIOPointStatus for status definitions).

*Size*, an integer value specifying the number of elements in the array pointer to be *pList*.

## Returns

$\geq 0$ , number of elements stored in the list

-1, if an error occurred

# SgGetOutputStatusList

## Prototype

```
int SgGetOutputStatusList(USHORT *pList, int Size)
```

## Overview

SgGetOutputStatusList stores the current status of all defined outputs (or up to Size entries) in the array pointed to by pList. The order of status words in the array corresponds to the order of outputs as returned by SgGetDefinedOutputs.

## Parameters

*pList*, a pointer to an array of unsigned short integers which hold the output status (see SgGetIOPointStatus for status definitions).

*Size*, an integer value specifying the number of elements in the array pointer to be *pList*.

## Returns

$\geq 0$ , number of elements stored in the list

-1, if an error occurred

# Login Functions

# SgLogin

## Prototype

int SgLogin(const char \*name, const char \*password, const char \*location)

## Overview

SgLogin validates the user name and password and logs the operator in. A valid user login must occur before any other commands can be issued

## Parameters

*name*, a pointer to the user name.

*password*, a pointer to the user password

*location*, a pointer to a character string indicating the location the user is logging in from.

## Returns

SG\_FAILURE, if not successful.

SG\_SUCCESS, if successful.

# SgLogout

## Prototype

int SgLogout(void)

## Overview

SgLogout logs the user out of Sage.

## Parameters

*none*

## Returns

SG\_FAILURE, if not logged in.  
SG\_SUCCESS, if successful.

# Reader Configuration Functions

## Reader Configuration Definitions and Structures

Card readers whether they are mag stripe, proximity or touch memory based support several different operational modes. Sage provides functions for retrieving and setting these modes using the reader configuration structure defined below.

```
// reader configuration parameters
typedef struct rdrparmtag {
    BOOL prealert;
    BOOL doorinout;
    BOOL interlock;
    BOOL systemcode;
    BOOL dualcust;
    BOOL powerlock;
    BOOL lockout;
    BOOL timedapb;
    BOOL localapb;
    BOOL globalapb;
    BOOL exitbutton;
    BOOL openclose;
    BOOL accessgrant;
    int unlocktime;
    int heldopentime;
    int prealerttime;
    int apbtime;
    int pinmode;
    int pinalg;

    int pinretry;
    int pinfromrange;
    int pintorange;
    int apbfromlevel;
    int apbtollevel;
} RDRPARMS;
```

### **prealert**

The prealert boolean value is true if door pre-alert mode is enabled and false otherwise.

### **doorinout**

The doorinout boolean value is true if both readers on a CP30/40 control a common lock and false otherwise.

### **interlock**

The interlock boolean value is true if both readers on a CP30/40 are operating in interlock mode and false otherwise.

**systemcode**

The systemcode boolean value is true if the reader is in system code mode and false otherwise.

**dualcust**

The dualcust boolean value is true if the reader is in dual custody mode and false otherwise.

**powerlock**

The powerlock boolean value is true if the reader is in power lock on request to exit mode and false otherwise.

**lockout**

The lockout boolean value is true if the reader is in lockout mode and false otherwise.

**timedapb**

The timedapb boolean value is true if the reader is in timed apb mode and false otherwise.

**localapb**

The localapb boolean value is true if the reader is in local apb mode and false otherwise.

**globalapb**

The globalapb boolean value is true if the reader is in global apb mode and false otherwise.

**exitbutton**

The exitbutton boolean value is true if the reader will audit request to exit button operation and false otherwise.

**openclose**

The openclose boolean value is true if the reader will audit door contact state changes and false otherwise.

**accessgrant**

The accessgrant boolean value is true if the reader will audit access granted events and false otherwise.

**unlocktime**

This value holds the readers unlock duration (in seconds) for access granted and momentary unlock commands. Valid range is from 1 to 60.

**heldopentime**

This value holds the readers door held open duration before going into alarm interval in seconds. Valid range is 10 to 300.

**prealerttime**

This value holds the percentage of the heldopentime interval before the reader sounds the door held open pre-alert alarm.

**apbtime**

This value holds the readers timed anti-passback value in minutes. Valid range is from 4 to 60.

**pinmode**

The pinmode field specifies the operation of attached AKP100 key pad devices. Valid values are 0 for card only, 1 for card + pin, and 2 for pin only operation.

**pinretry**

The pinretry value specifies the of attempts a user is allowed to make entering their pin correctly before the key pad reports an invalid pin event.

**pinfromrange, pinttorange**

These values define a range of card numbers that will be exempt from pin entry.

**apbfromlevel**

The global anti-passback level of this reader.

**apbtollevel**

The global anti-passback level this reader leads to.

# SgLoadReaderConfig

## Prototype

```
int SgLoadReaderConfig(int Id, RDRPARMS *Ptr)
```

## Overview

SgLoadReaderConfig loads the current reader configuration parameters into the buffer pointed to by *Ptr*. See Reader Configuration Definitions and Structures for the definition of RDRPARMS.

## Parameters

*Id*, card reader id number.

*Ptr*, a pointer to the parameter buffer.

## Returns

SG\_BAD\_PARAMETER, if the reader is not defined.

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_CONNECTION, if the server is not available.

SG\_SUCCESS, if the parameters were successfully loaded.

# SgSaveReaderConfig

## Prototype

```
int SgSaveReaderConfig(int Id, RDRPARMS *Ptr)
```

## Overview

SgSaveReaderConfig saves the reader configuration parameters stored in the buffer pointed to by *Ptr*. See Reader Configuration Definitions and Structures for the definition of RDRPARMS.

## Parameters

*Id*, card reader id number.

*Ptr*, a pointer to the parameter buffer.

## Returns

SG\_BAD\_PARAMETER, if the reader is not defined.

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_CONNECTION, if the server is not available.

SG\_SUCCESS, if the parameters were successfully saved.

## Card Reader Functions

# SgUnlockReader

## Prototype

```
int SgUnlockReader(USHORT *list, int count)
```

## Overview

SgUnlockReader submits an unlock command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid reader identification numbers.  
*count*, the number of reader identification numbers present in the list.

## Returns

SG\_BAD\_PARAMETER, if any of the reader numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

## SgUnlockMReader

### Prototype

```
int SgUnlockMReader(USHORT *list, int count)
```

### Overview

SgUnlockMReader submits a momentary unlock command to the Maestro server.

### Parameters

*list*, a pointer to an array of short integers containing valid reader identification numbers.  
*count*, the number of reader identification numbers present in the list.

### Returns

SG\_BAD\_PARAMETER, if any of the reader numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgRelockReader

## Prototype

int SgRelockReader(USHORT \*list, int count)

## Overview

SgRelockReader submits an re-lock command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid reader identification numbers.  
*count*, the number of reader identification numbers present in the list.

## Returns

SG\_BAD\_PARAMETER, if any of the reader numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgLockoutReader

## Prototype

```
int SgLockoutReader(USHORT *list, int count)
```

## Overview

SgLockoutReader submits a lock-out command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid reader identification numbers.  
*count*, the number of reader identification numbers present in the list.

## Returns

SG\_BAD\_PARAMETER, if any of the reader numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgRemoveLockoutReader

## Prototype

```
int SgRemoveLockoutReader(USHORT *list, int count)
```

## Overview

SgRemoveLockoutReader submits a lock-out off command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid reader identification numbers.  
*count*, the number of reader identification numbers present in the list.

## Returns

SG\_BAD\_PARAMETER, if any of the reader numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgLockElevatorReader

## Prototype

int SgLockElevatorReader(int id, int floor)

## Overview

SgLockElevatorReader submits a floor lock command to the Maestro server.

## Parameters

*id*, the logical id number of the elevator reader.

*floor*, the relay number of the floor to lock. This value ranges from 1 to either 16 or 64 depending on how the controller has been configured.

## Returns

SG\_BAD\_PARAMETER, if the reader id is invalid, or does not reference an elevator reader, or if the floor number is out of range. The command will not be performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgUnlockElevatorReader

## Prototype

```
int SgUnlockElevatorReader(int id, int floor)
```

## Overview

SgUnlockElevatorReader submits a floor unlock command to the Maestro server.

## Parameters

*id*, the logical id number of the elevator reader.

*floor*, the relay number of the floor to unlock. This value ranges from 1 to either 16 or 64 depending on how the controller has been configured.

## Returns

SG\_BAD\_PARAMETER, if the reader id is invalid, or does not reference an elevator reader, or if the floor number is out of range. The command will not be performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# Input Point Functions

# SgShuntInput

## Prototype

```
int SgShuntInput(USHORT *list, int count)
```

## Overview

SgShuntInput submits a global shunt command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid input identification numbers.  
*count*, the number of input identification numbers present in the list.

## Returns

SG\_BAD\_PARAMETER, if any of the input numbers are invalid and the command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgShuntMInput

## Prototype

```
int SgShuntMInput(USHORT *list, int count, int duration)
```

## Overview

SgShuntMInput submits a global momentary shunt command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid input identification numbers.

*count*, the number of input identification numbers present in the list.

*duration*, the shunt duration in seconds (1 to 32767).

## Returns

SG\_BAD\_PARAMETER, if any of the input numbers are invalid and the command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgUnshuntInput

## Prototype

```
int SgUnshuntInput(USHORT *list, int count)
```

## Overview

SgUnshuntInput submits a global remove shunt command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid input identification numbers.  
*count*, the number of input identification numbers present in the list.

## Returns

SG\_BAD\_PARAMETER, if any of the input numbers are invalid and the command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# Output Point Functions

# SgActivateOutput

## Prototype

```
int SgActivateOutput(USHORT *list, int count)
```

## Overview

SgActivateOutput submits an output activate command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid output identification numbers.  
*count*, the number of output identification numbers present in the list.

## Returns

SG\_BAD\_PARAMETER, if any of the output numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

# SgActivateMOutput

## Prototype

```
int SgActivateMOutput(USHORT *list, int count, int duration)
```

## Overview

SgActivateOutput submits a momentary output activation command to the Maestro server.

## Parameters

*list*, a pointer to an array of short integers containing valid output identification numbers.  
*count*, the number of output identification numbers present in the list.  
*duration*, the activation interval in seconds (1 to 99).

## Returns

SG\_BAD\_PARAMETER, if any of the output numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

## SgDeactivateOutput

### Prototype

```
int SgDeactivateOutput(USHORT *list, int count)
```

### Overview

SgDeactivateOutput submits an output de-activate command to the Maestro server.

### Parameters

*list*, a pointer to an array of short integers containing valid output identification numbers.  
*count*, the number of output identification numbers present in the list.

### Returns

SG\_BAD\_PARAMETER, if any of the output numbers are invalid. The command is not performed.

SG\_FAILURE, if the required memory for the command can not be allocated.

SG\_SUCCESS, if the command was submitted successfully.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if no connection is present with the server.

## **Access Profile Functions**

## Access Profile Structure

The Access Profile database consists of one or more records with the following structure;

```
// access profile record  
typedef struct accproftag {  
    char    in_use;  
    char    name[NAME_LGTH + 1];  
    char    rdr_schedules[MAX_READERS];  
    int     partition;  
} PROFILE;
```

### **in\_use**

The *in\_use* field is a reserved structure element and signifies whether the profile is deleted or not.

### **name**

This field identifies the access profile, and is referenced by the profile field of a personnel record (see Personnel Record Structure).

### **rdr\_schedules**

This character array consists of 512 (MAX\_READERS) entries, each containing an access schedule number. Schedule numbers are restricted to the range 0 through 15. The table is accessed by using the reader's id number - 1, to set or get the access schedule associated with that card reader. See the *Id* definition in the section Enumerating Defined Card Readers for a description of card reader id numbers.

### **partition**

This field is not currently used and it's use is reserved by Cansec.

# SgGetAccessProfile

## Prototype

```
int SgGetAccessProfile(char *name, PROFILE *buffer)
```

## Overview

SgGetAccessProfile retrieves the access profile identified by *name* and stores it in *buffer*.

## Parameters

*name*, a pointer to the string identifying the profile to retrieve.

*buffer*, a data buffer of type PROFILE to store the retrieved access profile.

## Returns

SG\_FAILURE, if the profile was not found or an error occurred.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if the access profile was retrieved successfully.

# SgAddAccessProfile

## Prototype

int SgAddAccessProfile(PROFILE \*buffer)

## Overview

SgAddAccessProfile adds a new access profile to the database.

## Parameters

*buffer*, a data buffer of type PROFILE containing the access profile to add.

## Returns

SG\_FAILURE, if an error occurred while adding the profile.

SG\_EXISTS, if an access profile already exists with the profile's name.

SG\_BAD\_PARAMETER, if one or more of the access schedules is not within the range 0 to 15.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if the access profile was added successfully.

# SgUpdateAccessProfile

## Prototype

```
int SgUpdateAccessProfile(PROFILE *buffer)
```

## Overview

SgUpdateAccessProfile updates the access profile with the data stored in *buffer*. If one or more cards are referencing the updated profile, the Maestro server will download the newly programmed schedules at the affected readers.

## Parameters

*buffer*, a data buffer of type PROFILE containing the profile data to store.

## Returns

SG\_BAD\_PARAMETER, if one or more of the access schedules is not within the range 0 to 15.

SG\_FAILURE, if the profile was not found or an error occurred.

    SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if the profile was retrieved successfully.

# SgDeleteAccessProfile

## Prototype

```
int SgDeleteAccessProfile(const char *name, const char *new)
```

## Overview

SgdeleteAccessProfile deletes the access profile referenced by name, converts existing users of the access profile to the new one specified by new, and updates the control panels appropriately.

## Parameters

*name*, a pointer to the name of the profile to delete.

*new*, a pointer to the name of a profile to convert existing users of *name* to.

## Returns

SG\_NOT\_FOUND if either profile does not exist, or the names are the same.

SG\_BAD\_PARAMETER, if an attempt is made to delete "No Access".

SG\_FAILURE, if the profile was not found or an error occurred.

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_SUCCESS, if the profile was retrieved successfully.

## SgOpenProfileDatabase

### Prototype

```
int SgOpenProfileDatabase(void)
```

### Overview

SgOpenProfileDatabase opens the profile database for subsequent read operations.

### Parameters

*none*

### Returns

$\geq 0$ , a handle for use in subsequent database read operations  
 $< 0$ , an error occurred and the open has failed.

## SgCloseProfileDatabase

### Prototype

```
int SgCloseProfileDatabase(int handle)
```

### Overview

SgCloseProfileDatabase closes a previously opened handle to the profile database.

### Parameters

*handle*, a valid handle returned from SgOpenProfileDatabase.

### Returns

SG\_BAD\_HANDLE, if the handle is not open, or is out of range.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if the handle was closed successfully.

# SgGetFirstAccessProfile

## Prototype

```
int SgGetFirstAccessProfile(int handle, PROFILE *buffer)
```

## Overview

SgGetFirstAccessProfile retrieves the first defined access profile in the database.

## Parameters

*handle*, a valid handle returned from SgOpenProfileDatabase.

*buffer*, a data buffer of type PROFILE containing the profile data to store.

## Returns

SG\_NOT\_FOUND, if the profile database does not exist.

SG\_FAILURE, if an error occurred while opening or reading the database.

SG\_BAD\_HANDLE, if the handle is not open, or is out of range.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if the profile was retrieved successfully.

## SgGetNextAccessProfile

### Prototype

```
int SgGetNextAccessProfile(int handle, PROFILE *buffer)
```

### Overview

SgGetNextAccessProfile retrieves the next defined access profile.

### Parameters

*handle*, a valid handle returned from SgOpenProfileDatabase.

*buffer*, a data buffer of type PROFILE containing the profile data to store.

### Returns

SG\_NOT\_FOUND, if no other profiles are defined following *name*.

SG\_FAILURE, if an error occurred while opening or reading the database.

SG\_BAD\_HANDLE, if the handle is not open, or is out of range.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if the profile was retrieved successfully.

## SgGetAccessProfileCount

### Prototype

```
int SgGetAccessProfileCount(void)
```

### Overview

SgGetAccessProfileCount returns the number of defined access profiles.

### Parameters

*none*

### Returns

-1 if an error occurred, otherwise the number of defined profiles.

## SgGetDefinedAccessProfiles

### Prototype

```
int SgGetDefinedAccessProfiles(SGLISTDEF *pList, int Size)
```

### Overview

SgGetDefinedAccessProfiles builds and retrieves a list of all defined access profiles. The Id and Optional fields of the SGLISTDEF structure are not used. See SGLISTDEF Structure for additional information.

### Parameters

*pList*, a pointer to an array of SGLISTDEF records;

*Size*, the number of records pointed to by pList.

### Returns

$\geq 0$ , the number of records stored.

-1, an error occurred

# Personnel Database Functions

## Personnel Record Structure

The personnel or card holder database consists of one or more records of the structure presented below.

```
typedef struct card {
    char    in_use;
    char    id[CARD_ID_LGTH + 1];
    char    name[NAME_LGTH + 1];
    WHEN    validate_date;
    WHEN    void_date;
    char    status;
    char    user1[USER_LGTH + 1];
    char    user2[USER_LGTH + 1];
    char    user3[USER_LGTH + 1];
    char    user4[USER_LGTH + 1];
    char    user5[USER_LGTH + 1];
    char    user6[USER_LGTH + 1];
    char    user7[USER_LGTH + 1];
    char    user8[USER_LGTH + 1];
    char    user9[USER_LGTH + 1];
    char    user10[USER_LGTH + 1];
    char    notes[NOTES_LGTH + 1];
    char    profile[NAME_LGTH + 1];
    char    linkname[LNKNAME_LGTH + 1];
    BOOL    changed;
    char    photofile[PATH_LGTH + 1];
    char    signaturefile[PATH_LGTH + 1];
    int     handicapped;
    int     unlock_time;
    int     dho_time;
    int     partition;
} CARD;
```

### **in\_use**

The *in\_use* field is a reserved structure element and signifies whether the card is deleted or not.

### **id**

The id field represents the card number associated with this card holder. This is a 5 digit character field ranging from 1 to 19500 (or 40000).

### **name**

This character array stores the card holders name.

### **validate\_date**

This structure is used for validating cards at some future date. By default, it is set to the date a card holder's record is first added.

**void\_date**

This structure is used for voiding a card at some future date. By default it is set to January 1<sup>st</sup>, 2050.

**status**

The *status* field indicates the current disposition of the card. It may be set to any one of the following defined constants.

- ACTIVE**           The card is currently active, using a valid access profile.
- INACTIVE**       The card is currently inactive and is using a *No Access* profile.
- LOST**            The card is lost and is set to *No Access*.
- STOLEN**         The card has been stolen and is set to *No Access*.
- DESTROYED**    The card has been destroyed and is set to *No Access*.
- PENDING**       The card is pending validation.
- PATIENT**        The card holder is considered a patient (Maestro Patient Wander systems only).

**user1 - user10**

These are discretionary data fields which may contain any alpha-numeric data.

**notes**

The notes field is a discretionary field which may contain any alpha-numeric data.

**profile**

This field contains the access profile assigned to this card. Maestro creates the *No Access* profile when it first starts. This profile designates no access at any reader in the system. Cards with a status of *INACTIVE*, *LOST*, *STOLEN*, or *DESTROYED* must use the *No Access* profile. Cards which are *ACTIVE* or *PENDING* must not use the *No Access* profile.

**linkname**

This field contains a photo badge design name. It's use is reserved by Cansec.

**changed**

This boolean field is true if a new photo has been captured, but a new badge has not been printed.

**photofile**

This field contains the name of the associated image file stored in the Images directory. It's use is reserved by Cansec.

**signaturefile**

This field is not currently used and it's use is reserved by Cansec.

**handicapped**

This integer field is set to 1 if the cardholder is considered to be handicapped and 0 otherwise. If set to 1, a reader access granted will result in door and door held open times to be extended.

**unlock\_time**

If *handicapped* is set to 1, the door unlock time will be extended by the value entered here. Default value for this field is 7 and may range up to 60.

**dho\_time**

If *handicapped* is set to 1, the door held open time will be extended by the value entered here.

**partition**

This field is not currently used and its use is reserved by Cansec.

The database is indexed, and may be accessed in ascending order by opening it using one of the following field identifiers.

<b>ID_NDX</b>	Card id number field
<b>NAME_NDX</b>	Name field
<b>USER1_NDX</b>	User field 1
<b>USER2_NDX</b>	User field 2
<b>USER3_NDX</b>	User field 3
<b>USER4_NDX</b>	User field 4
<b>USER5_NDX</b>	User field 5
<b>USER6_NDX</b>	User field 6
<b>USER7_NDX</b>	User field 7
<b>USER8_NDX</b>	User field 8
<b>USER9_NDX</b>	User field 9
<b>USER10_NDX</b>	User field 10
<b>PROF_NDX</b>	Access profile field

See SgOpenCardDatabase for information.

## SgInitCardRecord

### Prototype

```
int SgInitCardRecord(int nmb, CARD *ptr)
```

### Overview

SgInitCardRecord initializes the personnel record by setting the id field to the value passed in *nmb*, setting the validate date field to today's date, the void field to January 1<sup>st</sup>, 2050, the card status to *INACTIVE*, and the access profile to *No Access*. All other fields are set to null.

### Parameters

*nmb*, an integer representing the card id number of the personnel record.

*ptr*, a pointer to a buffer of type CARD to be initialized.

### Returns

SG\_BAD\_PARAMETER, if the card number is out of range.

SG\_SUCCESS, otherwise.

## SgGetCardRecord

### Prototype

```
int SgGetCardRecord(int nmb, CARD *ptr)
```

### Overview

SgGetCardRecord retrieves the personnel record identified by *nmb* and returns it in *ptr*.

### Parameters

*nmb*, an integer representing the card number of the personnel record to retrieve.  
*ptr*, a pointer to a buffer of type *CARD* which will hold the retrieved personnel record.

### Returns

SG\_BAD\_PARAMETER, if the card number is out of range.

SG\_NOT\_FOUND, if record is not present in the database.

SG\_FAILURE, if an error occurred.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful

# SgUpdateCardRecord

## Prototype

```
int SgUpdateCardRecord(int nibr, CARD *ptr)
```

## Overview

SgUpdateCardRecord updates the personnel database with the new data. It may also re-validate at the necessary control panels if the access profile has changed.

## Parameters

*nibr*, a integer representing the card number of the personnel record to update.  
*ptr*, a pointer to a buffer of type CARD which holds the updated personnel record.

## Returns

SG\_BAD\_PARAMETER, if the card number is out of range.  
SG\_PER\_VALIDATE\_DATE, if the validate field is out of range.  
SG\_PER\_VOID\_DATE, if the void field is out of range.  
SG\_PER\_STATUS, if the status field is invalid.  
SG\_PER\_PROFILE, if the profile does not exist.  
SG\_BAD\_RECORD, if other errors exist.  
SG\_NOT\_FOUND, if the record does not exist.  
SG\_FAILURE, if an error occurred.  
SG\_NO\_CONNECTION, if the server is not available to validate the record.  
    SG\_NO\_LOGIN, if an operator has not logged in to Sage  
SG\_SUCCESS, if successful.

# SgAddCardRecord

## Prototype

```
int SgAddCardRecord(int nمبر, CARD *ptr)
```

## Overview

SgAddCardRecord adds a new record to the personnel database and validates the card at the control panels.

## Parameters

*nمبر*, an integer representing the card number of the personnel record to add.

*ptr*, a pointer to a buffer of type CARD which holds the new personnel record.

## Returns

SG\_BAD\_PARAMETER, if the card number is out of range.

SG\_PER\_VALIDATE\_DATE, if the validate field is out of range.

SG\_PER\_VOID\_DATE, if the void field is out of range.

SG\_PER\_STATUS, if the status field is invalid.

SG\_PER\_PROFILE, if the profile does not exist.

SG\_BAD\_RECORD, if other errors exist.

SG\_EXISTS, if the record already exists.

SG\_FAILURE, if an error occurred.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_NO\_CONNECTION, if the server is not available to validate the record.

SG\_SUCCESS, if successful.

## SgVoidCardRecord

### Prototype

```
int SgVoidCardRecord(int nibr)
```

### Overview

SgVoidCardRecord, voids a card record at all the control panels and sets it's status to *INACTIVE* and it's access profile to *No Access*.

### Parameters

*nibr*, an integer representing the card number of the personnel record to void.

### Returns

SG\_BAD\_PARAMETER, if the card number is out of range.

SG\_NOT\_FOUND, if record is not present in the database.

SG\_FAILURE, if an error occurred.

SG\_NO\_CONNECTION, if the server is not available to void the record.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful

# SgDeleteCardRecord

## Prototype

int SgDeleteCardRecord(int nibr)

## Overview

SgDeleteCardRecord, voids a card record at all the control panels and deletes it from the personnel database.

## Parameters

*nibr*, an integer representing the card number of the personnel record to delete.

## Returns

SG\_BAD\_PARAMETER, if the card number is out of range.

SG\_NOT\_FOUND, if record is not present in the database.

SG\_FAILURE, if an error occurred.

SG\_NO\_CONNECTION, if the server is not available to void the record.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful

## SgOpenCardDatabase

### Prototype

```
int SgOpenCardDatabase(int indexfield)
```

### Overview

SgOpenCardDatabase opens the personnel database for subsequent read operations.

### Parameters

*Indexfield*, an identifier indicating the field by which to access the database. See Personnel Record Structure for field identifier names.

### Returns

$\geq 0$ , a handle for use in subsequent database read operations  
 $< 0$ , an error occurred and the open has failed.

## SgCloseCardDatabase

### Prototype

```
int SgCloseCardDatabase(int handle)
```

### Overview

SgCloseCardDatabase closes a previously opened handle to the personnel database.

### Parameters

*handle*, a valid handle returned from SgOpenCardDatabase.

### Returns

SG\_BAD\_HANDLE, if the handle is not open, or is out of range.  
SG\_SUCCESS, if the handle was closed successfully.

## SgGetFirstCardRecord

### Prototype

```
int SgGetFirstCardRecord(int handle, CARD *ptr)
```

### Overview

SgGetFirstCardRecord retrieves the first personnel record sorted by the index field specified in SgOpenCardDatabase.

### Parameters

*handle*, a valid handle returned from SgOpenCardDatabase.

*ptr*, a pointer to a buffer of type CARD which will hold the retrieved personnel record.

### Returns

SG\_NOT\_FOUND, if no records are present in the database.

SG\_FAILURE, if an error occurred.

SG\_BAD\_HANDLE, if the handle is not open, or is out of range.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful

# SgGetNextCardRecord

## Prototype

int SgGetNextCardRecord(int handle, CARD \*ptr)

## Overview

SgGetNextCardRecord retrieves the next personnel record from the personnel database, sorted by the index field specified in SgOpenCardDatabase.

## Parameters

*handle*, a valid handle returned from SgOpenCardDatabase.

*ptr*, a pointer to a buffer of type CARD which will hold the retrieved personnel record.

## Returns

SG\_BAD\_PARAMETER, if the card number is out of range.

SG\_NOT\_FOUND, if the next record is not present in the database.

SG\_FAILURE, if an error occurred.

SG\_BAD\_HANDLE, if the handle is not open, or is out of range.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful

# SgFindCardRecord

## Prototype

```
int SgFindCardRecord(int handle, char *key, CARD *ptr)
```

## Overview

SgFindCardRecord retrieves the first personnel record from the personnel database matching the search string *key*. The actual field that is searched for *key* is based on the field identifier specified in SgOpenCardDatabase. If a matching record can not be found, the record pointer is positioned at the beginning of the file.

## Parameters

*handle*, a valid handle returned from SgOpenCardDatabase.

*Key*, a null terminated search string to match. Comparisons begin at the beginning of the field.

*ptr*, a pointer to a buffer of type CARD which will hold the retrieved personnel record.

## Returns

SG\_FAILURE, if an error occurred.

SG\_BAD\_HANDLE, if the handle is not open, or is out of range.

SG\_NOT\_FOUND, if a matching record could not be found

    SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful

## SgGetCardRecordCount

### Prototype

```
int SgGetCardRecordCount(void)
```

### Overview

SgGetCardRecordCount returns the number of records present in the personnel database.

### Parameters

None

### Returns

-1 if an error occurred.  
Database record count otherwise.

## SgGetDefinedCards

### Prototype

```
int SgGetDefinedCards(SGLISTDEF *pList, int Size)
```

### Overview

SgGetDefinedCards builds and retrieves a list of all defined card holders in the personnel file. The Id field of the SGLISTDEF structure contains the card number, which may be used for later retrieval. The Optional field is not used. See SGLISTDEF Structure for additional information.

### Parameters

*pList*, a pointer to an array of SGLISTDEF records;  
*Size*, the number of records pointed to by pList.

### Returns

$\geq 0$ , the number of records stored.  
 $-1$ , an error occurred

# SgGetPersonnelTitles

## Prototype

```
int SgGetPersonnelTitles(TITLES *titles)
```

## Overview

SgGetPersonnelTitles reads the current personnel user field names into the buffer pointed to by titles.

## Parameters

*titles*, a pointer to a buffer of type TITLES. The definition for the TITLES structure is presented below.

```
// personnel title file structure  
typedef struct titlestag {  
    char   field1[USER_TITLE_LGTH + 1];    // user field 1 name  
    char   field2[USER_TITLE_LGTH + 1];    // user field 2 name  
    char   field3[USER_TITLE_LGTH + 1];    // user field 3 name  
    char   field4[USER_TITLE_LGTH + 1];    // user field 4 name  
    char   field5[USER_TITLE_LGTH + 1];    // user field 5 name  
    char   field6[USER_TITLE_LGTH + 1];    // user field 6 name  
    char   field7[USER_TITLE_LGTH + 1];    // user field 7 name  
    char   field8[USER_TITLE_LGTH + 1];    // user field 8 name  
    char   field9[USER_TITLE_LGTH + 1];    // user field 9 name  
    char   field10[USER_TITLE_LGTH + 1];   // user field 10 name  
} TITLES;
```

## Returns

SG\_FAILURE, if the operation failed.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful.

# SgPutPersonnelTitles

## Prototype

```
int SgPutPersonnelTitles(TITLES *titles)
```

## Overview

SgPutPersonnelTitles updates the current personnel-user field titles, with the data pointed by titles.

## Parameters

*titles*, a pointer to a buffer of type TITLES. See SgGetPersonnelTitles for a description of this structure.

## Returns

SG\_FAILURE, if the operation failed.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if successful.

# Group Functions

## Group Definitions and Structures

Maestro group definitions and structure are discussed in this section. Maestro supports four group types. Each of the group API entry points utilizes one of the following definitions to identify the type of group.

<b>READER_GROUP</b>	Card reader device group type.
<b>INPUT_GROUP</b>	Input point device group type.
<b>OUTPUT_GROUP</b>	Output point device group type.
<b>CARD_GROUP</b>	Card holder group type

The definition for a group is presented below. Each group may consist of up to 200 entries as defined by the `MAX_ITEMS_PER_GROUP` definition. Group members that are not defined are set to zero. Members in a group are sorted into ascending order, with the first zero member encountered indicating the end of defined members.

```
typedef struct grouptag {
    char    name[NAME_LGTH + 1];

    int     members[MAX_ITEMS_PER_GROUP];
} SGGROUP;
```

Sage provides a set of four functions for manipulating these groups. Each of these functions is displayed below.

```
int SgGroupLoad(int type, int which, SGGROUP *group);
```

```
int SgGroupSave(int type, int which, SGGROUP *group);
```

```
int SgGroupDelete(int type, int which);
```

# SgGroupAdd

## Prototype

```
int SgGroupAdd(int type, SGGROUP *group)
```

## Overview

SgGroupAdd adds a new group of type *type*.

## Parameters

*type*, one of the pre-defined types listed in the section Group Definitions and Structures.  
*group*, a pointer to the group member data.

## Returns

SG\_BAD\_PARAMETER, if the group type parameter is not valid.  
SG\_FAILURE, if Sage has not been initialized or the operation failed.  
    SG\_NO\_LOGIN, if an operator has not logged in to Sage  
SG\_SUCCESS, if group was successfully added.

# SgGroupSave

## Prototype

int SgGroupSave(int type, int which, SGGROUP \*group)

## Overview

SgGroupSave updates the specified group with new group data.

## Parameters

*type*, one of the pre-defined types listed in the section Group Definitions and Structures.

*which*, an integer representing the group number (1 to 2000).

*group*, a pointer to the group member data.

## Returns

SG\_BAD\_PARAMETER, if the group type or which parameters are not valid.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if group was successfully saved.

## SgGroupDelete

### Prototype

int SgGroupDelete(int type, int which)

### Overview

SgGroupDelete deletes the specified group.

### Parameters

*type*, one of the pre-defined types listed in the section Group Definitions and Structures.  
*which*, an integer representing the group number (1 to 2000).

### Returns

SG\_BAD\_PARAMETER, if the group type or which parameters are not valid.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if group was successfully deleted.

# SgGroupLoad

## Prototype

int SgGroupLoad(int type, int which, SGGROUP \*group)

## Overview

SgGroupLoad retrieves the specified group.

## Parameters

*type*, one of the pre-defined types listed in the section Group Definitions and Structures.

*which*, an integer representing the group number (1 to 2000).

*group*, a pointer to the group member data buffer.

## Returns

SG\_BAD\_PARAMETER, if the group type or which parameters are not valid.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_LOGIN, if an operator has not logged in to Sage

SG\_SUCCESS, if group was successfully retrieved.

# SgGetDefinedGroups

## Prototype

```
int SgGetDefinedGroups(int Type, SGLISTDEF *pList, int Size)
```

## Overview

SgGetDefinedGroups builds and retrieves a list of all defined groups of a given type. The *Id* field of the SGLISTDEF structure contains the record number, which may be used for later retrieval. The *Optional* field contains the group type `READER_GROUP`, `INPUT_GROUP`, `OUTPUT_GROUP`, or `CARD_GROUP`. See SGLISTDEF Structure for additional information.

## Parameters

*Type*, an integer value specifying the type of group to enumerate.  
*pList*, a pointer to an array of SGLISTDEF records;  
*Size*, the number of records pointed to by pList.

## Returns

$\geq 0$ , the number of records stored.  
 $-1$ , an error occurred

# Audit Trail Functions

## SgOpenAuditTrail

### Prototype

int SgOpenAuditTrail(WHEN \*Start, WHEN \*End)

### Overview

SgOpenAuditTrail, returns a handle to the Maestro audit trail based on the date and time ranges specified..

### Parameters

*Start*, the starting date and time of the audit trail range to include.

*End*, the ending date and time of the audit trail range to include.

### Returns

-1, if an error occurred.

A handle to be used in subsequent calls otherwise.

## SgCloseAuditTrail

### **Prototype**

```
void SgCloseAuditTrail(int Handle)
```

### **Overview**

SgCloseAuditTrail closes a previously opened audit trail handle.

### **Parameters**

*Handle*, a previously opened audit trail handle.

### **Returns**

None

# SgReadAuditTrail

## Prototype

BOOL SgReadAuditTrail(int Handle, SGEVMSG \*Buffer)

## Overview

SgReadAuditTrail reads the next record from the audit trail file.

## Parameters

*Handle*, a previously opened audit trail handle.

*Buffer*, a pointer to a buffer of type SGEVMSG to hold the retrieved record.

## Returns

TRUE if successful

FALSE otherwise

# Schedule Functions

## Schedule Structure

All Maestro schedules are based on an 8 day schedule (Sunday through Saturday, plus a holiday) broken into 48 half hour periods per day. The include file Sage.h provides definitions for periods and days as below.

```
// schedule day definitions
#define SUNDAY 0
#define MONDAY 1
#define TUESDAY 2
#define WEDNESDAY 3
#define THURSDAY 4
#define FRIDAY 5
#define SATURDAY 6
#define HOLIDAY 7
#define MAX_DAYS 8

// schedule definitions
#define MAX_PERIODS (24 * 2) // number of 1/2 hour schedule
periods
#define MAX_SYS_SCHED 32 // max system timezone schedules
#define MAX_OUTPUT_SCHED 15 // max Output timezone schedules
#define MAX_INPUT_SCHED 32 // max input shunt timezone schedules
#define MAX_ELEV_SCHED 15 // max elevator unlock schedules
#define MAX_ACC_SCHED 15 // max access schedules
#define MAX_UNLK_SCHED 50 // max unlock schedules
```

A single schedule is an array of 8, single day records where each record consists of 48 intervals.

```
// schedule definition
typedef struct schedule {
    char name[NAME_LGTH + 1];
    char periods[MAX_DAYS][MAX_PERIODS];
} SCHEDULE;
```

The name field of the SCHEDULE definition is not currently used and is reserved for future use by Cansec Systems.

# SgGetAccessSchedule

## Prototype

```
int SgGetAccessSchedule(SCHEDULE *ptr)
```

## Overview

SgGetAccessSchedule retrieves the 15 defined access schedules into the schedule buffer *ptr*.

## Parameters

*ptr*, a pointer to an array of MAX\_ACC\_SCHEDULE records of type SCHEDULE..

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_FAILURE, if the command failed and the schedules were not loaded.  
SG\_SUCCESS, if the command was successful.

# SgUpdateAccessSchedule

## Prototype

```
int SgUpdateAccessSchedule(SCHEDULE *ptr)
```

## Overview

SgUpdateAccessSchedule saves the 15 access schedules pointed to by the pointer *ptr* to disk and downloads the changes to all control panels.

## Parameters

*ptr*, a pointer to an array of MAX\_ACC\_SCHEDULE records of type SCHEDULE.

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_NO\_CONNECTION, if there is no connection with the Maestro server.  
SG\_FAILURE, if the command failed and the schedules were not updated.  
SG\_SUCCESS, if the command was successful.

# SgGetUnlockSchedule

## Prototype

```
int SgGetUnlockSchedule(SCHEDULE *ptr)
```

## Overview

SgGetUnlockSchedule retrieves the 50 defined reader unlock schedules into the schedule buffer *ptr*.

## Parameters

*ptr*, a pointer to an array of MAX\_UNLK\_SCHEDULE records of type SCHEDULE..

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_FAILURE, if the command failed and the schedules were not loaded.  
SG\_SUCCESS, if the command was successful.

# SgUpdateUnlockSchedule

## Prototype

```
int SgUpdateUnlockSchedule(SCHEDULE *ptr)
```

## Overview

SgUpdateUnlockSchedule saves the 50 unlock schedules pointed to by the pointer *ptr* to disk and downloads the changes to all control panels.

## Parameters

*ptr*, a pointer to an array of MAX\_UNLK\_SCHEDULE records of type SCHEDULE.

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_NO\_CONNECTION, if there is no connection with the Maestro server.  
SG\_FAILURE, if the command failed and the schedules were not updated.  
SG\_SUCCESS, if the command was successful.

# SgGetElevUnlockSchedule

## Prototype

```
int SgGetElevUnlockSchedule(SCHEDULE *ptr)
```

## Overview

SgGetElevUnlockSchedule retrieves the 15 defined elevator unlock schedules into the schedule buffer *ptr*.

## Parameters

*ptr*, a pointer to an array of MAX\_ELEV\_SCHEDULE records of type SCHEDULE..

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_FAILURE, if the command failed and the schedules were not loaded.  
SG\_SUCCESS, if the command was successful.

# SgUpdateElevUnlockSchedule

## Prototype

```
int SgUpdateElevUnlockSchedule(SCHEDULE *ptr)
```

## Overview

SgUpdateElevUnlockSchedule saves the 15 elevator unlock schedules pointed to by the pointer *ptr* to disk and downloads the changes to all control panels.

## Parameters

*ptr*, a pointer to an array of MAX\_ELEV\_SCHEDULE records of type SCHEDULE.

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_NO\_CONNECTION, if there is no connection with the Maestro server.  
SG\_FAILURE, if the command failed and the schedules were not updated.  
SG\_SUCCESS, if the command was successful.

# SgGetShuntSchedule

## Prototype

```
int SgGetShuntSchedule(SCHEDULE *ptr)
```

## Overview

SgGetShuntSchedule retrieves the 32 defined input shunt schedules into the schedule buffer *ptr*.

## Parameters

*ptr*, a pointer to an array of MAX\_INPUT\_SCHEDULE records of type SCHEDULE..

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_FAILURE, if the command failed and the schedules were not loaded.  
SG\_SUCCESS, if the command was successful.

# SgUpdateShuntSchedule

## Prototype

```
int SgUpdateShuntSchedule(SCHEDULE *ptr)
```

## Overview

SgUpdateShuntSchedule saves the 32 Shunt schedules pointed to by the pointer *ptr* to disk and notifies all Maestro clients that they have been changed.

## Parameters

*ptr*, a pointer to an array of MAX\_INPUT\_SCHEDULE records of type SCHEDULE.

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_NO\_CONNECTION, if there is no connection with the Maestro server.  
SG\_FAILURE, if the command failed and the schedules were not updated.  
SG\_SUCCESS, if the command was successful.

# SgGetOutputSchedule

## Prototype

```
int SgGetOutputSchedule(SCHEDULE *ptr)
```

## Overview

SgGetOutputSchedule retrieves the 15 defined output schedules into the schedule buffer *ptr*.

## Parameters

*ptr*, a pointer to an array of MAX\_OUTPUT\_SCHEDULE records of type SCHEDULE..

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_FAILURE, if the command failed and the schedules were not loaded.  
SG\_SUCCESS, if the command was successful.

# SgUpdateOutputSchedule

## Prototype

```
int SgUpdateOutputSchedule(SCHEDULE *ptr)
```

## Overview

SgUpdateOutputSchedule saves the 15 output schedules pointed to by the pointer *ptr* to disk and downloads the changes to all control panels.

## Parameters

*ptr*, a pointer to an array of MAX\_OUTPUT\_SCHEDULE records of type SCHEDULE.

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_NO\_CONNECTION, if there is no connection with the Maestro server.  
SG\_FAILURE, if the command failed and the schedules were not updated.  
SG\_SUCCESS, if the command was successful.

# SgGetSystemSchedule

## Prototype

```
int SgGetSystemSchedule(SCHEDULE *ptr)
```

## Overview

SgGetSystemSchedule retrieves the 32 general system schedules into the schedule buffer *ptr*.

## Parameters

*ptr*, a pointer to an array of MAX\_SYSTEM\_SCHEDULE records of type SCHEDULE..

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_FAILURE, if the command failed and the schedules were not loaded.  
SG\_SUCCESS, if the command was successful.

# SgUpdateSystemSchedule

## Prototype

```
int SgUpdateSystemSchedule(SCHEDULE *ptr)
```

## Overview

SgUpdateSystemSchedule saves the 32 general system schedules pointed to by the pointer *ptr* to disk and notifies all Maestro clients that they have been changed.

## Parameters

*ptr*, a pointer to an array of MAX\_SYSTEM\_SCHEDULE records of type SCHEDULE.

## Returns

SG\_NO\_LOGIN, if a valid operator login command has not taken place.  
SG\_NO\_CONNECTION, if there is no connection with the Maestro server.  
SG\_FAILURE, if the command failed and the schedules were not updated.  
SG\_SUCCESS, if the command was successful.

# Control Profile Functions

## Control Profile Structure

Control profiles are data structures, which are executed by a Maestro server in response to an outside device event. Each profile consists of 10 records, which specify an event and an action to take if the event occurs. Three type of control profiles are defined, one for each device type, and distinguishable by one of the definitions (see Sage.h);

READER\_PROFILE  
INPUT\_PROFILE  
OUTPUT\_PROFILE

For each profile type, there is a corresponding set of event types which can server as triggers for a subsequent action. Definitions for the event types are presented below.

### Card Reader Events (READER\_PROFILE)

RDREV_NONE	Default event
RDREV_GRANTED	Access granted event
RDREV_DENIED	Access Denied event
RDREV_DURESS	Reader duress event
RDREV_HANDICAP	Handicapped user event
RDREV_GRP_AG	Access granted event by a member of a card group
RDREV_GRP_AD	Access denied event by a member of a card group
RDREV_FORCED	Forced entry event
RDREV_DHO	Door held open event
RDREV_TAMPER	Card reader tamper event
RDREV_UNLOCK	Card reader unlocked event
RDREV_OFFLINE	Card reader offline event

### Input Point Events (INPUT\_PROFILE)

INEV_NONE	Default event
INEV_SECURE	Input point has returned to secure
INEV_ALARM	Input point has gone into alarm
INEV_TROUBLE	Input point has gone into trouble
INEV_OFFLINE	Input point has gone offline

### Output Point Events (OUTPUT\_PROFILE)

OUTEV_NONE	Default event
OUTEV_ON	Output point has been activated
OUTEV_OFF	Output point has been de-activated
OUTEV_OFFLINE	Output point has gone offline

If a device state change should match one the events above, the type of action to take is determined by one of the object type definitions displayed below.

OBJECT_READER	The action will apply to a single card reader
---------------	---

OBJECT_INPUT	The action will apply to a single input point
OBJECT_OUTPUT	The action will apply to a single output point
OBJECT_READER_GRP	The action will apply to a group of card readers
OBJECT_INPUT_GRP	The action will apply to a group of input points
OBJECT_OUTPUT_GRP	The action will apply to a group of output points

Finally, the type of action to take is determined from one of action definitions;

ACTION_ON	Unlock, Shunt, or Activate
ACTION_OFF	Relock, Remove Shunt, or Deactivate
ACTION_MOMENTARY	Momentary Unlock, Shunt, or Activate

The LINK structure contains this logic through its definition below;

```
// control profile structure for events and actions
typedef struct linktag {
    short intevent;           // reader, input, or output event type
    short intevent_grp;      // optional group number
    short intobject_type;    // type object OBJECT_READER etc.
    short intobjectid;      // objects id number
    short intaction;        // type of action to take ACTION_ON etc.
    short intduration;      // momentary actions duration
} LINK;
```

A second related type of link concerns events and the execution of macros. The structure definitions follow;

```
// control profile structure for events and macros
typedef struct maclinktag {
    short int event;         // reader, input, or output event type
    short intevent_grp;     // optional group number
    char macroname[NAME_LGTH + 1]; // macro filename
} MACLINK;
```

The PGMREC structure organizes this event data into a structure that can be associated with one of more devices.

```
// control profile record structure
typedef struct pgmtag {
    BOOL    inuse;           // reserved
    int     partition;      // reserved
    int     type;           // one of READER_PROFILE,
etc.
    int     control_schedule; // system schedule
    char    name[NAME_LGTH + 1]; // profile name
```

```
LINK          events[LINKS_PER_OBJECT]; // up to 10 events
MACLINK      macevents[LINKS_PER_OBJECT]; // up to 10 events
} PGMREC;
```

## SgAddCtrlProfile

### Prototype

```
int SgAddCtrlProfile(PGMREC *buffer)
```

### Overview

SgAddCtrlProfile adds the control profile stored in *buffer*.

### Parameters

*buffer*, a data buffer of type PGMREC containing the new control profile.

### Returns

> 0, index of the newly added profile  
-1 if an error occurred

# SgGetCtrlProfile

## Prototype

int SgGetCtrlProfile(int which, PGMREC \*buffer)

## Overview

SgGetCtrlProfile retrieves the control profile specified by *which* (1 to MAX\_PGM\_RECORDS) and stores it in *buffer*.

## Parameters

*which*, an integer value from 1 to MAX\_PGM\_RECORDS specifying the record to retrieve.

*buffer*, a data buffer of type PGMREC to store the retrieved control profile.

## Returns

SG\_FAILURE, if an error occurred

SG\_NOT\_FOUND if record undefined

SG\_BAD\_PARAMETER, if index out of range

SG\_NO\_LOGIN if an operator has not logged in to Sage

SG\_SUCCESS, if the control profile was retrieved successfully.

## SgUpdateCtrlProfile

### Prototype

int SgUpdateCtrlProfile(int which, PGMREC \*buffer)

### Overview

SgUpdateCtrlProfile updates the control profile specified by *which* (1 to MAX\_PGM\_RECORDS) with the data stored in *buffer*.

### Parameters

*which*, an integer value from 1 to MAX\_PGM\_RECORDS specifying the record to retrieve.

*buffer*, a data buffer of type PGMREC containing the new control profile data.

### Returns

SG\_FAILURE, if an error occurred

SG\_BAD\_PARAMETER, if index out of range

SG\_NO\_LOGIN if an operator has not logged in to Sage

SG\_SUCCESS, if the control profile was retrieved successfully.

## SgDeleteCtrlProfile

### Prototype

int SgDeleteCtrlProfile(int which)

### Overview

SgDeleteCtrlProfile deletes the control profile specified by which (1 to MAX\_PGM\_RECORDS).

### Parameters

*which*, an integer value from 1 to MAX\_PGM\_RECORDS specifying the record to delete.

### Returns

SG\_FAILURE, if an error occurred  
SG\_BAD\_PARAMETER, if index out of range  
SG\_NO\_LOGIN if an operator has not logged in to Sage  
SG\_SUCCESS, if the control profile was retrieved successfully.

## SgGetDefinedCtrlProfiles

### Prototype

```
int SgGetDefinedCtrlProfiles(SGLISTDEF *pList, int Size)
```

### Overview

SgGetDefinedCtrlProfiles builds and retrieves a list of all defined control profiles. The *Id* field of the SGLISTDEF structure contains the record number, which may be used for later retrieval. The *Optional* field contains the profile type `READER_PROFILE`, `INPUT_PROFILE`, or `OUTPUT_PROFILE`. See SGLISTDEF Structure for additional information.

### Parameters

*pList*, a pointer to an array of SGLISTDEF records;  
*Size*, the number of records pointed to by pList.

### Returns

$\geq 0$ , the number of records stored.  
-1, an error occurred

# Association Functions

## Association Definitions and Structures

Maestro creates links between data objects and devices using what are termed associations. Associations are nothing more than integer arrays where the array index - 1 specifies the device id and the data value at that index position specifies an object record number. Sage supports six association types. Each of the association API entry points uses one of the following definitions to identify the type of association.

<b>RDRCTRL</b>	Card Reader/Control Profile Association
<b>INCTRL</b>	Input Point/Control Profile Association
<b>OUTCTRL</b>	Output Point/Control Profile Association
<b>RDRSCHED</b>	Card Reader/Unlock Schedule Association
<b>INSCHED</b>	Input Point/Shunt Schedule Association
<b>OUTSCHED</b>	Output Point/Activation Schedule Association

The definition for an association is presented below. Each element is merely a short integer which refers to the record number of the appropriate data object. For example, the value 3 at position 10 in the RDRSCHED associations, indicates that card reader 11 is associated with unlock schedule 3.

```
// association record definition
typedef struct assoctag {
    short intindex;
} ASSOC;
```

Sage provides a two functions for manipulating these associations. These functions are displayed below.

```
int SgLoadAssociations(int Type, ASSOC *Ptr, int Size);
```

```
int SgSaveAssociations(int Type, ASSOC *Ptr, int Size);
```

# SgLoadAssociations

## Prototype

int SgLoadAssociations(int Type, ASSOC \*Ptr, int Size)

## Overview

SgLoadAssociations retrieves the specified association array.

## Parameters

*Type*, one of the pre-defined types listed in the section Association Definitions and Structures.

*Ptr*, a pointer to the association buffer.

*Size*, an integer value indicating the size of the array.

## Returns

SG\_BAD\_PARAMETER, if the association type was not valid.

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_SUCCESS, if the associations were successfully retrieved.

## SgSaveAssociations

### Prototype

```
int SgSaveAssociations(int Type, ASSOC *Ptr, int Size)
```

### Overview

SgSaveAssociations saves the specified association array.

### Parameters

*Type*, one of the pre-defined types listed in the section Association Definitions and Structures.

*Ptr*, a pointer to the association buffer.

*Size*, an integer value indicating the size of the array.

### Returns

SG\_BAD\_PARAMETER, if the association type was not valid.

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_SUCCESS, if the associations were successfully saved.

# Miscellaneous Functions

# SgGetDefinedMacros

## Prototype

```
int SgGetDefinedMacros(SGLISTDEF *pList, int Size)
```

## Overview

SgGetDefinedMacros builds and retrieves a list of all defined macros. The Id and Optional fields of the SGLISTDEF structure are not used. See SGLISTDEF Structure for additional information.

## Parameters

*pList*, a pointer to an array of SGLISTDEF records;

*Size*, the number of records pointed to by pList.

## Returns

$\geq 0$ , the number of records stored.

-1, an error occurred

# SgLogError

## Prototype

```
int SgLogError(char *msg, char *file, int line)
```

## Overview

SgLogError permits the Sage application to record events in the Sage log file. While the application is running, Sage records program exceptions to an ASCII log file based on the application's name (see SgInitialize for specifying the application name). The application may also submit messages to the log using SgLogError.

## Parameters

*msg*, a pointer to a null terminated message to log.

*file*, a null terminated string designating the source module name. See the `__FILE__` preprocessor directive in your compiler documentation.

*line*, a integer value indicating the line number of the source module. See the `__LINE__` preprocessor directive in your compiler documentation.

## Returns

SG\_FAILURE, if an error occurred.

SG\_SUCCESS, if successful

# SglsSysTZone

## Prototype

BOOL SglsSysTZone(int schedule)

## Overview

SglsSysTZone queries the system schedule identified by *schedule* to determine if it is currently operating within the boundaries of an active time zone.

## Parameters

*schedule*, a integer value specifying one of the 32 system schedules 1 through 32. Schedule 0 is always FALSE.

## Returns

TRUE, if the system if the system schedule identified by *schedule* is currently operating within the boundaries of an active time zone.

FALSE otherwise.

# SgGlobalAPB

## Prototype

int SgGlobalAPB(BOOL State)

## Overview

SgGlobalAPB enables or disables global anti-passback processing based on the boolean value *State*.

## Parameters

*State*, pass TRUE to enable global anti-passback or FALSE to disable it.

## Returns

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_CONNECTION, if the server is not available.

SG\_SUCCESS, if the function completed successfully.

## SgGlobalAPBState

### Prototype

```
int SgGlobalAPBState(BOOL *State)
```

### Overview

SgGlobalAPBState sets the boolean value pointed to be *State* to true if global anti-passback is currently enabled and false otherwise.

### Parameters

*State*, a pointer to a boolean value which will be set to the current global anti-passback state.

### Returns

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_CONNECTION, if the server is not available.

SG\_SUCCESS, if the function completed successfully.

# SgDateTime

## Prototype

int SgDateTime(void)

## Overview

SgDateTime downloads the current date and time to all control panels.

## Parameters

*none*

## Returns

SG\_NO\_LOGIN, if an operator has not logged in to Sage.

SG\_FAILURE, if Sage has not been initialized or the operation failed.

SG\_NO\_CONNECTION, if the server is not available.

SG\_SUCCESS, if the function completed successfully.

## SgRestoreComms

### Prototype

```
int SgRestoreComms(void)
```

### Overview

SgRestoreComms re-starts communications with control panels which have timed out.

### Parameters

*none.*

### Returns

SG\_NO\_LOGIN, if an operator has not logged in to Sage.  
SG\_NO\_CONNECTION, if the server is not available.  
SG\_SUCCESS, if the function completed successfully.

# Index

## A

Application Guidelines .....	13
Association Definitions and Structures .....	212

## C

Control Profile Structure .....	204
---------------------------------	-----

## N

New For Version 2.0 .....	7
New For Version 2.1 .....	7

## R

Reader Configuration Definitions and Structures .....	125
---	-----

## S

Schedule Structure .....	189
SgAddCtrlProfile .....	206
SgCloseCardDatabase .....	169
SgCloseProfileDatabase .....	153
SgDateTime .....	221
SgDeleteAccessProfile .....	151
SgDeleteCtrlProfile .....	209
SgGetAccessProfileCount .....	156
SgGetAccessSchedule .....	190
SgGetCtrlProfile .....	207
SgGetDefinedAccessProfiles .....	157
SgGetDefinedCards .....	174
SgGetDefinedCtrlProfiles .....	210
SgGetDefinedGroups .....	183
SgGetDefinedMacros .....	216
SgGetElevUnlockSchedule .....	194
SgGetFirstAccessProfile .....	154
SgGetInputStatusList .....	119
SgGetLocalName .....	81
SgGetNextAccessProfile .....	155
SgGetOutputSchedule .....	198
SgGetOutputStatus .....	120
SgGetOutputStatusList .....	120
SgGetPersonnelTitles .....	175
SgGetReaderStatusList .....	117
SgGetRunTimeRecord .....	77
SgGetShuntSchedule .....	196
SgGetSystemSchedule .....	200
SgGetUnlockSchedule .....	192
SgGetWorkingDirectory .....	76
SgGlobalAPB .....	219
SgGlobalAPBState .....	220

SgInitCardRecord .....	162
SgIsSysTZone .....	218
SGLISTDEF Structure .....	38
SgLoadAssociations .....	213
SgLoadReaderConfig .....	128
SgLockElevatorReader.....	136
SgLogin.....	122
SgLogout .....	123
SgOpenCardDatabase .....	168
SgOpenProfileDatabase.....	152
SgPutPersonnelTitles .....	176
SgRestoreComms .....	222
SgSaveAssociations .....	214
SgSaveReaderConfig.....	129
SgSetMsgMask.....	113
SgUnlockElevatorReader .....	137
SgUpdateAccessSchedule .....	191
SgUpdateCtrlProfile .....	208
SgUpdateElevUnlockSchedule.....	195
SgUpdateOutputSchedule.....	199
SgUpdateShuntSchedule .....	197
SgUpdateSystemSchedule.....	201
SgUpdateUnlockSchedule.....	193

## W

Whats New .....	7
-----------------	---