SENTINEL SuperPro™

SentinelSuperPro

Programmer’s Reference Guide

RAINBOW TECHNOLOGIES

CONFIDENTIAL INFORMATION

The SentinelSuperPro security system is designed to protect your software products from unauthorized use. The less information that unauthorized people have regarding your security system, the greater your protection. It is in your best interest to protect the information herein from access by unauthorized individuals. Please read the Developer's Agreement at the beginning of this document for safeguarding requirements.

Part Number 700496-001 Revision A

Software Releases SP-5.1 and later

System Driver Releases PD-5.36 and later

INTERNET: //www.rainbow.com

RAINBOW TECHNOLOGIES, INC.
50 Technology Drive, Irvine, CA 92618
TEL: (949) 450-7300, (800) 852-8569 *Fax: (949) 450-7450*

RAINBOW TECHNOLOGIES LTD.
4 The Forum, Hanworth Lane, Chertsey, Surrey KT16 9JX, United Kingdom
TEL: (44) 1 932 579200 *Fax: (44) 1 932 570743*

RAINBOW MICROPHAR
122, Avenue Charles de Gaulle, 92522 Neuilly-sur-Seine Cedex, France
TEL: (33) 1 41 43 29 00 *Fax: (33) 1 46 24 76 91*

RAINBOW TECHNOLOGIES GMBH
Lise Meitner Strasse 1, 85716 Unterschleissheim, Germany
TEL: (49) 89 32 17 98 0 *Fax: (49) 89 32 17 98 50*

SOFTWARE LICENSE AND DEVELOPER’S AGREEMENT

All Products (including developer’s kits, Sentinel hardware keys, diskettes or other magnetic media, software, documentation and all future orders) are subject to the terms stated below. If you disagree with these terms, please return the Product and the documentation to Rainbow, postage prepaid, within three days of your receipt, and Rainbow will provide you with a refund, less freight and normal handling charges.

1. You may not copy or reproduce all or any part of the Product, except as authorized in item 2 below. Removal, emulation or reverse-engineering of all or any part of the Product constitutes an unauthorized modification to the Product and is specifically prohibited. Nothing in this license permits you to derive the source code of the software files that Rainbow has provided to you. Your software programs must be protected or licensed using a licensed and registered copy of this Rainbow Product. Rainbow provides no other warranty to any person, other than the Limited Warranty provided to the original purchaser of this Product.

2. a. You may make archival copies of the software files and you may modify and merge them into your software programs for the sole purpose of implementing the Product to protect and/or license your programs according to the Rainbow documentation provided with the Product. All software files remain Rainbow’s exclusive property.

   b. Rainbow’s Sentinel System Driver Software and other Rainbow software files listed in the “Licensee Redistribution Allowances” section (if it is defined in the Product’s documentation) may be copied and distributed to your customers for the sole purpose of executing your protected or licensed software programs according to the Rainbow documentation provided with the Product.

   c. No license is granted to Licensee to sell, license, distribute, market or otherwise dispose of any software files or other component of the Product except when embedded in your software programs. Copies of your software programs must bear a valid copyright notice and must be distributed such that the object code for the Product cannot be extracted.

3. Rainbow warrants the Product and the magnetic media on which the software files are provided to be substantially free from significant defects in materials and workmanship under normal use for a period of twelve (12) months from the date of delivery of the Product to you. In the event of a claim under this warranty, Rainbow’s sole obligation is to replace or repair, at Rainbow’s option, any Product free of charge. Any replaced parts shall become Rainbow’s property.

4. Warranty claims must be made in writing during the warranty period and within seven (7) days of the observation of the defect, accompanied by evidence satisfactory to Rainbow. Prior to returning any Product to Rainbow, you must obtain a Return Merchandise Authorization (RMA) number and shipping instruction from Rainbow. Products returned to Rainbow shall be shipped with freight and insurance paid.

5. Except as stated above, there is NO OTHER WARRANTY, REPRESENTATION, OR CONDITION REGARDING RAINBOW'S PRODUCTS, SERVICES, OR PERFORMANCE, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Rainbow is not responsible for any delays beyond its control. Rainbow's entire liability for damages to you or any other party for any cause whatsoever, whether in contract or in tort, including negligence, shall not exceed the price you paid for the unit of Product that caused the damages or that are the subject matter of, or are directly related to, the cause of action. In no event will Rainbow be liable for any damages caused by your failure to perform your obligations, or for any loss of data, profits, savings, or any other consequential and incidental damages, or for any claims by you based on any third-party claim.

Licensee Redistribution Allowances

The SentinelSuperPro Licensees may release the Sentinel System Driver diskette for installation with their Sentinel-protected application.

FCC Notice To USERS

This equipment has been tested and found to comply with the limits for a Class B digital device, pursuant to Part 15 of the FCC Rules. Operation is subject to the following conditions: (1) this device may not cause harmful interference, and (2) this device must accept any interference received, including interference that may cause undesired operation.

This equipment generates, uses, and can radiate frequency energy and, if not installed and used in accordance with the instructions, may cause harmful interference to radio communications. However, there is no guarantee that interference will not occur in a particular installation.

If interference problems do occur, please consult the system equipment owner's manual for suggestions. Some of these suggestions include relocation of the computer system away from the television or radio, or placing the computer AC power connection on a different circuit or outlet.

Change or modifications to this product without the express approval of Rainbow Technologies, Inc., could result in non-FCC compliance, and void the user's authority to operate this equipment.

International Quality Standard Certification

Rainbow Technologies, Inc. Irvine CA facility has been issued the ISO 9002 Certification, the globally recognized standard for quality, by British Standards Institution as of December 1994.

Certificate Number FM 30128

European Community Directive Conformance Statement

This product is in conformity with the protection requirements of EC Council Directive 89/336/EEC. Conformity is declared to the following applicable standards for electro-magnetic compatibility immunity and susceptibility; CISPR22 and IEC801. This product satisfies the CLASS B limits of EN 55022.

Table of Contents

About This Manual

This manual explains how to protect an application with the SentinelSuperPro software protection system. It covers the following topics:

- How SentinelSuperPro memory cells can be programmed.
- The client interface routines used to communicate with the SentinelSuperPro.
- How to use the SentinelWizard’s Advanced Editor.
- The techniques used to protect linked-in drivers with encryption when not in use.
- General techniques for protecting an application, with examples that demonstrate how to implement common protection strategies.
- Error codes.
- Quick reference charts you may find helpful when planning your protection strategy and programming your keys.

About the Developer’s Guide

Please refer to the *SentinelSuperPro Developer’s Guide* for the following topics:

- Installing the SentinelSuperPro software.
- Attaching the hardware key.
- Running the SentinelWizard.
- Running the SentinelShell.
- Protecting and running your application.
- Choosing the correct driver.
- Troubleshooting, problem reporting, shipping, and handling.
- Glossary of terms.

Conventions Used in this manual

The following documentation conventions are used in this manual:

Hexadecimal values

Unless otherwise noted, all values are expressed in hexadecimal format.

Modulo operations

The syntax “n MOD m” is used to indicate any number that produces *n* as a remainder when divided by *m*. For example, 3 MOD 8 represents any number that, when divided by 8, yields a remainder of 3. Therefore, 3 MOD 8 represents the hex numbers 3, 0B, 13, 1B, 23, 2B, and so on.

Additional Sources of Information

Every SentinelSuperPro security system includes README files. These files contain the latest information on the system.

See the *SentinelSuperPro Developer’s Guide* for a complete list of Technical Support services.

Chapter 1 - Introduction to the SentinelSuperPro

The SentinelSuperPro is a hardware/software protection system that secures applications from unauthorized use. If the correct hardware key is not attached to the computer, the protected application is not fully functional. Therefore, only legitimate customers can use your product.

To implement a protection scheme, you program your application to send calls to the hardware key to verify its presence. The frequency of these calls, and the action taken if no key is found, are left to your discretion.

SentinelSuperPro keys are customized for each developer. Therefore, other developers cannot reprogram your protected application's key. Also, possession of one developer's key does not allow a user to run another developer's protected application.

Once installed, the SentinelSuperPro is a transparent part of your software package. No additional action is required of the user.

System Components

The SentinelSuperPro system consists of three major components:

- The **hardware key** is a programmable, read/write memory device that provides the responses required to unlock your application. Each key contains 64 memory cells, 56 of which are programmable. Memory cells can be programmed with data values to provide fixed responses or serve as counters. Each key also contains internal logic that transforms data based on encryption strings you define.
- The **Application Program Interface** (API) is a set of functions used to access the SentinelSuperPro security system. The API manages the communication between your application, the SentinelSuperPro driver, and the hardware key.

- The **SentinelWizard**, a Windows utility, lets you program the hardware key's memory cells. These programmed cells provide the values your application uses to determine whether or not the key is attached to the PC. The SentinelWizard also generates pseudo-code you can use to add API calls to your application.

Language/Operating System Support

The list of languages and compilers supported by the SentinelSuperPro system changes frequently. For the most current list, see the README file or the Rainbow Technologies homepage on the Internet (//www.rainbow.com).

Support is provided for applications running in a variety of operating systems (including DOS, Windows 3.x, Windows 95/98, Windows NT, and OS/2) on a wide range of hardware platforms (including Intel, DEC Alpha, Power PC, NEC, and FMR).

To support applications running under various operating environments, Rainbow provides driver software that can be configured at installation time. For example, a DOS application may be used by your customer in a Windows or Windows NT DOS box. The driver software can be incorporated into your installation program to provide consolidated support.

Chapter 2 - Feature Overview

The SentinelSuperPro protection system provides the following major features:

- The ability to protect your application from unauthorized use through software locks.
- The ability to program a single key in a variety of different ways to meet the needs of your application.
- A two-level password protection scheme that secures the programming process.
- A means for the end user to activate an inactive application in the field.
- A means to control the execution of demos and trial purchases.
- The ability to reprogram memory cells in the field.
- The ability to use one key to protect more than one application.
- The ability to encrypt the driver so that it is effectively “hidden” between calls to the hardware key.
- An enhanced algorithm engine option for increased protection.

These features are described in detail in this chapter. Instructions for implementing each feature are provided in later chapters of this book.

Protection from Piracy

To use the SentinelSuperPro to protect your program from piracy, you insert a series of “software locks” into your application. Each lock is a call to a SentinelSuperPro API routine, and requires the presence of the hardware key in order to succeed. If the key is missing, an error code is returned to your program so the unauthorized copy will know to terminate. Software can be copied illegally, but pirated software will not run.

Flexible Programming

A primary advantage of the SentinelSuperPro system is that one key can be programmed to provide many diverse types of both fixed and variable responses. This gives you tremendous variations in the types of software locks you can create.

For example, cells can be used to store fixed data such as serial numbers, user names, or codes that control feature access. Such data can be read to simply verify that the key is still attached. You can also use stored data to control program flow or application functions. Data words can be defined as read-only (locked) or read/write.

Cells can also store encryption strings (called “algorithm descriptors”) that are used to scramble input strings sent by your application. Other cells can be programmed as counters used to restrict the number of executions.

The first eight cells in each key are reserved for system information. With some restrictions, the other 56 cells can be used in any way you desire.

Password Protection

The ability to program SentinelSuperPro units is protected by two passwords:

- Your **write password** lets you write to undefined cells and read/write data words.
- Your **overwrite password** lets you write to all other non-restricted cells: read-only data words, counters, and algorithm words.

Cells 0 through 7 are restricted and cannot be reprogrammed, even with the overwrite password.

You must have your passwords in order to program keys with the SentinelWizard. You must also include the passwords in your protected application to reprogram cells in the field. (The passwords are required parameters for some API calls.)

The overwrite password is extremely powerful and should be guarded carefully. It is recommended that you do not compromise the password by using it in your released application. To avoid doing so, design your protection scheme so that cells that require this password are set at the factory and do not change in the field.

Note: If your protection scheme does not require the overwrite password from your application, Rainbow can program each of your keys with a “random overwrite password.” Each key will contain a different overwrite key password and can not be changed in the field.

Algorithm Activation

Algorithm activation is a protection technique you can use to ensure that only legitimate users can use your application. Basically, you ship your application in an unusable state, and provide a mechanism for legitimate purchasers to activate it.

To use this technique, you define an algorithm in your hardware key as “inactive.” This makes it unavailable for queries, which means that a query sent from your application receives an invalid response. (A “query” is an API call in which your application sends a data string, the string is scrambled according to the key’s internal logic and an algorithm descriptor, and the scrambled string is returned to your application.)

To activate an inactive algorithm, the customer must enter a password you have previously defined. If you are using the SentinelSuperPro Manufacturing Utility, the utility will create the password for you. The USAFE activation utility can then be run by the end user and in conjunction with DSAFE, the password will be re-created and used to activate the product.

Generally, there are two situations in which you might want to use algorithm activation:

- You set the algorithm descriptor to “inactive” at your factory. The user must contact you for the correct algorithm password in order to run the program for the first time. If you use one key to protect multiple applications, you would issue the password that activates the program the customer purchased.
- You set the algorithm descriptor to “active” at your factory, and define an associated counter. Your application decrements the counter each time it runs. Once the counter reaches 0, the algorithm descriptor becomes inactive, causing further queries to fail. The user must contact you for the algorithm password to use the application again.

Demo Program Control

You can program the SentinelSuperPro to count the number of times a program is executed. This is especially useful for demonstration software.

To count executions, you program a memory cell as a counter, and set the initial value. Each time the application is executed, it decrements the counter by 1. When the counter reaches 0, you code your application to prevent it from running again.

The usual way in which demo control is performed is by linking the counter cell to an algorithm descriptor that is used in queries. When the counter reaches 0, the algorithm is deactivated automatically. Subsequent queries return incorrect response values. You code your application to stop running if an incorrect value is returned.

If you use the SentinelShell system to protect your software, you can also limit demo programs by time and number of days. For example, your application can be enabled for a period of 30 days. For details on the SentinelShell, see the *SentinelSuperPro Developer's Guide*.

The counter/algorithm combination may also be linked to a password. If the customer purchases an extension of the software, you provide him or her with the activation password. Let the SuperPro Manufacturing Utility create the password originally, use USAFE/DSAFE to recreate the password and then activate the product in the field.

Reprogramming Cells in the Field

Of special significance is the option for your protected application to dynamically reprogram the SentinelSuperPro in the field. For example, you can write installation programs that ask the user for values such as passwords. The entered values are then written to the hardware key and can be used in later software locks.

For added security when reprogramming keys in the field, use the SentinelSuperPro in conjunction with Rainbow's SentinelSAFE system. For more information about the SentinelSAFE, contact your sales representative.

Multiple Applications Sharing One Key

More than one application can use the same hardware key for protection. Usually, this is implemented by assigning certain cells to each application. This gives each application its own algorithm descriptor(s) to use for software locks.

Customers who run several of your applications on the same computer need to attach *only one key* to protect them all. This saves space and improves the PC's appearance in the customer's work area.

See **Using One Key for Multiple Applications** in Chapter 7 for a sample programming layout in which seven programs share a single key.

Driver Encryption

For an extra level of protection against potential hackers, the SentinelSuperPro linked-in driver file can be encrypted when it is not in use. This process allows your program to have an "invisible" copy of the driver on disk and in memory until the driver code is actually utilized.

You can customize the encryption process by choosing various parameters, such as which encryption method to use and which part of the driver is to be encrypted. You link your application to the encrypted driver file, then call routines from within your application to decrypt the driver before communicating with the hardware key. Optionally, the driver can be re-encrypted between calls.

By linking an encrypted version of the SentinelSuperPro driver into your program, you get the benefit of a "drop in" driver for communicating with the hardware key. Also, the driver code in your application program becomes extremely difficult for a hacker to interpret.

Note: Driver encryption is available for linked-in drivers only. It is not supported for system drivers or DLLs.

Enhanced Algorithm Engine

You can choose the type of protection used for each algorithm: the original algorithm engine for normal protection, or the enhanced engine, the most powerful form of protection. When you program keys with the SentinelWizard, you can easily activate the enhanced algorithm engine.

The enhanced engine requires the latest model of the SentinelSuperPro.

Note: For maximum security, use of the enhanced algorithm engine is recommended for all algorithms.

Chapter 3 - Programmable Memory

Every SentinelSuperPro key contains 128 bytes of memory organized as 64 words of 16 bits each. The words (cells) are addressed as locations 0 through 3F hex, and can be addressed randomly to provide the responses required by your software. Cells 0 through 7 in each key are restricted. You can program cells 8 through 3F hex.

When you program a cell, you assign it various attributes. These attributes determine how the cell can be used by your application.

The information in this chapter provides a basis for understanding how the SentinelSuperPro key can be used. You should read this chapter before programming the key, and should refer to it when designing your protection strategy.

Overview and Terminology

Each cell you program can be one of the following general types:

- A **data word** can store data such as customer information, serial numbers, passwords, and check digits. You code your application to read the cell and evaluate (and act upon) the stored value. A data word cell can be programmed as read-only or as read/write.
- A **counter word** contains an initial value you set and your application decrements. A typical use of a counter is to limit the number of times a demonstration program can be executed.
- An **algorithm descriptor** contains a bit pattern that defines how the hardware key is to scramble a data string sent by your application. The key uses your algorithm descriptor plus an internally stored, proprietary algorithm to transform the input string. You design your application to send data strings to the key and evaluate (and act upon) the values returned.

  All algorithm descriptors are two cells long. Algorithm descriptors may have activation passwords and deactivation counters associated with them.

To define how you want to use a particular cell, you assign it a code called a **cell type**. The cell type classifies the type of data stored in the cell; this in turn affects how the cell can be used. Each cell type is identified by a two-letter abbreviation. For example, CW identifies a counter word. Certain cell types are designed to be used in groups; for example, algorithm descriptors may have counters and passwords linked to them.

Every cell type has an **access code** associated with it. This code controls how the cell can be used by your application (for example, read-only or read/write). Access codes are numbers from 0 through 3.

Some cell types have **address restrictions**. Such cell types can be assigned only to specific memory cells.

In addition to the cell type, you program a **cell value** into each cell. Each cell can contain a 16-bit hex value (0000 through FFFF).

Algorithm descriptors can be active or inactive. The **active/inactive bit** in the cell value you program controls whether or not the algorithm descriptor is active and therefore able to be used in a query.

Another bit in the cell value determines whether the **enhanced algorithm engine** is activated for this algorithm descriptor.

Restricted Cells

Cells 0 through 7 in each key are restricted cells that contain fixed, pre-programmed system information. These cells are described in Table 3-1.

Table 3-1. Restricted Cells

Cell(s)	Contents	Readable?
0	Serial number, sequentially assigned per key *	Yes
1	Developer ID; unique to your company/product	Yes
2	Overwrite password word 1	No
3	Overwrite password word 2	No
4	Write password	No
5-7	Reserved for use by Rainbow Technologies	No

* *Maximum 16-bit value (1 - 65535). Serial number will repeat. If you require unique serial numbers, please contact your Rainbow representative. Rainbow must program the keys.*

The write and overwrite passwords stored in cells 2, 3, and 4 must be used by your application to perform certain functions. The passwords are an additional form of protection against tampering.

Access Codes

Every cell type has an access code. The access code controls how a cell of that type can be used by your application. For example, some cell types have an access code that permits the values to be both read and overwritten. Others are read-only, and some are not even readable.

When you program a cell with the Advanced Editor, you do not assign the access code. The Wizard determines the access code based on the protection feature you are implementing. If your application programs or reprograms cells in the field, it must specify the new access code.

Table 3-2 describes all access codes.

Table 3-2. Access Codes

Code	Description

Code	Description
0	Read/write data word Your application can read the cell and, if the write password is supplied, modify its contents.
1	Read-only (locked) data word Your application can read the cell but cannot change it without the overwrite password.
2	Counter word The cell contains a value that your application can decrement using the write password. The cell's value cannot be changed (other than by decrementing it) without the overwrite password.
3	Locked and hidden/algorithm word Your application cannot read the cell's value. Modification requires the overwrite password. The cells are hidden (unreadable).

Cell Values

In general, any cell can contain any hex value from 0000 through FFFF. The only restriction is that the value in the second word of an algorithm descriptor controls (1) whether the algorithm is active or inactive, and (2) whether the enhanced algorithm engine is enabled or disabled. For more information, see **Special Rules for Algorithm Values** in this chapter.

Cell Types

Cell types are summarized in Table 3-3 and described in detail on the following pages. Some of the descriptions refer to API functions; these are described in Chapter 4.

Table 3-3. Cell Types

Cell Types

Cell Type	Access Code	Description
**	0	Undefined
AA	3	Active Algorithm
AH	3	Algorithm Half
AP	3	Algorithm Password
CA	2	Algorithm Counter Word
CW	2	Counter Word
DI	1	Developer ID
DL	1	Locked Data Word
DW	0	Data Word
IA	3	Inactive Algorithm
OP	3	Overwrite Password
RW	3	Reserved Word
SN	1	Serial Number
WP	3	Write Password

** (Undefined)

The Undefined cell type is used to identify a cell that has not yet been programmed or that is not required in your protection strategy. The Undefined cell type is identified by two asterisks (**).

Note: Cells you do not require for your protection strategy can be left undefined. Alternatively, you may wish to program unused cells as read-only data words or algorithm/hidden words. This prevents them from being accessed, and also makes them available if you decide to expand your protection scheme in the future.

Access Code

An Undefined cell has an access code of 0 (read/write data).

Valid Addresses

Any unrestricted cell (08-3F) can be classified as Undefined.

AA (Active Algorithm)

The Active Algorithm (AA) cell type defines an active (enabled) algorithm descriptor. An algorithm descriptor consists of two adjacent cells with access codes of 3. The values in these cells affect the way an input string is scrambled via the *sproQuery()* API function. An algorithm descriptor must be active in order to be used for a query.

The value in the second AA word must be between 8000 and FFFF. For details, see **Special Rules for Algorithm Values**.

AA cells can have a password and counter(s) associated with them.

Access Code

An AA cell type has an access code of 3 (algorithm/hidden).

Valid Addresses

The first AA word must be at an unrestricted, even address (0 MOD 2). Additional restrictions apply if a counter and/or password is used. For details, see **Valid Addresses for Algorithms**.

AH (Algorithm Half)

The Algorithm Half (AH) cell type can be used for each of the two words required for an algorithm descriptor. The algorithm descriptor created by two AH cells is basically the same as that created by two AA or IA cells. The difference is that you can program the descriptor in two steps, which may be useful in some protection schemes.

The value in the second AH word must be between 0000 and 7FFF (for an inactive algorithm) or 8000 and FFFF (for an active algorithm). For more information, see **Special Rules for Algorithm Values**.

AH cells can have a password and counter(s) associated with them.

Note: Use of the AH cell type requires a thorough understanding of algorithm descriptors. Consider using the AA and IA cell types instead.

Access Code

The AH cell type has an access code of 3 (algorithm/hidden).

Valid Addresses

An AH cell can be located in any unrestricted cell (08-3F). You must leave an adjacent cell vacant for the other half of the algorithm descriptor. Also, the first AH word of the pair must be an even-numbered cell.

Additional address restrictions apply if a counter and/or password is used. For details, see **Valid Addresses for Algorithms**.

AP (Algorithm Password)

The Algorithm Password (AP) cell type is used to “turn on” an inactive algorithm descriptor so that it can be used for queries. This technique allows activation of an algorithm descriptor at a customer’s site. For a complete description of this feature, see **Using Activation Passwords** in Chapter 7.

The AP cell type must be two words long and must immediately follow the algorithm descriptor it activates.

Access Code

The AP cell type has an access code of 3 (algorithm/hidden). It cannot be directly read or written to; its value is used only to verify a user-supplied password during execution of the *sproActivateAlgorithm()* API function.

Because an AP cell has an access code of 3, it can be used as an algorithm descriptor. See **Querying Activation Passwords** in Chapter 7 for details.

Valid Addresses

The AP cell type must be located immediately after a two-word algorithm descriptor (cell type AA, AH, or IA). The two algorithm descriptor words must be at addresses equal to 0 MOD 4 and 1 MOD 4. Therefore, the two AP words must be at locations equal to 2 MOD 4 and 3 MOD 4.

Additional restrictions apply if a counter is also associated with the algorithm. For details, see **Valid Addresses for Algorithms**.

CA (Algorithm Counter Word)

The Algorithm Counter Word (CA) cell type defines a counter that deactivates an associated algorithm descriptor when the counter reaches 0. You program an initial value into the counter, then decrement it using the *sproDecrement()* API function.

The CA cell type must immediately precede the algorithm descriptor it deactivates.

This cell type can be used to control the number of times an application can be executed. For more information, see **Controlling Demo Applications** in Chapter 7.

Optionally, you can associate two counters (two CA cells) with one algorithm descriptor. In this case, the first counter to reach 0 deactivates the algorithm descriptor. If desired, you could use the second counter after the algorithm is reactivated with a password.

Access Code

The CA cell type has an access code of 2 (counter). It can be read but cannot be written to except by the *sproDecrement()* function.

Valid Addresses

A CA cell is always located immediately before a two-word algorithm descriptor (cell type AA, AH, or IA). The first word of this algorithm descriptor must be at a location equal to 4 MOD 8. Therefore, the CA counter must be located at an address equal to 3 MOD 8.

If you use two algorithm counters, the first must be at a location equal to 2 MOD 8, and the second must be at a location equal to 3 MOD 8.

For more information on valid algorithm locations, see **Valid Addresses for Algorithms**.

CW (Counter Word)

The Counter Word (CW) cell type is used for a counter that is *not* used to deactivate an algorithm descriptor. You program an initial value into the counter word, then decrement it using the *sproDecrement()* API function. You code your application to check the value in the counter and proceed accordingly if the value reaches 0.

Access Code

The CW cell type has an access code of 2 (counter). It can be read but cannot be written to except by the *sproDecrement()* function.

Valid Addresses

A CW word can be located at any unrestricted cell (08-3F).

Note: If you program a counter in an address equal to 3 MOD 8, and you use the next two cells for an algorithm descriptor, the counter will function as an algorithm counter. When the counter reaches 0, the algorithm will be deactivated, even if you did not intend for that to happen.

DI (Developer ID)

The Developer ID (DI) cell type is used for cell 1 only. This is a read-only data word that contains the unique developer ID assigned to you by Rainbow Technologies or your distributor. You cannot assign cell type DI to any cell.

Access Code

The DI word has an access code of 1 (locked). You can read the developer ID but cannot change it.

Valid Addresses

The only word that can be defined as cell type DI is cell 1.

DL (Locked Data Word)

The Locked Data Word (DL) cell type is used for data words you want your application to read but not write to.

Access Code

A DL cell has an access code of 1 (locked). After you program the cell, your application can read it but cannot change it without the overwrite password.

Valid Addresses

A DL cell can be located at any unrestricted address (08-3F).

DW (Data Word)

The Data Word (DW) cell type can store any value you wish to use in your software protection scheme. This value can be read and/or changed by your application. It can also be decremented.

Access Code

A DW cell has an access code of 0 (read/write). It can be reprogrammed using the write password.

Valid Addresses

A DW cell can be located at any unrestricted address (08-3F).

IA (Inactive Algorithm)

The Inactive Algorithm (IA) cell type defines an inactive (disabled) algorithm descriptor. An algorithm descriptor consists of two adjacent cells with access codes of 3. The values in these cells affect the way an input string is scrambled via the *sproQuery()* function. An inactive algorithm descriptor cannot be used for a query until it is activated by the *sproActivateAlgorithm()* function.

The value in the second IA word must be between 0000 and 7FFF. See **Special Rules for Algorithm Values** for more information.

IA cells should always have a password associated with them, so the algorithm can be activated. They can also have one or two counters.

Access Code

An IA cell type has an access code of 3 (algorithm/hidden).

Valid Addresses

The first IA word must be at an unrestricted, even address (0 MOD 2). Additional restrictions apply if a counter and/or password is used. For details, see **Valid Addresses for Algorithms**.

OP (Overwrite Password)

The Overwrite Password (OP) cell type is used for cells 2 and 3 only. These cells contain the overwrite password assigned to you by Rainbow Technologies or your distributor. This value is pre-programmed into your key and cannot be modified.

You must use your overwrite password to program, change, or delete the value in any cell that has an access code of 1, 2, or 3 (that is, any cell that is not read/write data or undefined).

Access Code

The OP cell type has an access code of 3 (algorithm/hidden).

Valid Addresses

The only cells defined as type OP are cells 2 and 3.

RW (Reserved Word)

The Reserved Word (RW) cell type is used for cells 5 through 7 only. These are hidden words that are reserved for use by Rainbow Technologies. You cannot assign cell type RW to any cell.

Access Code

An RW word has an access code of 3 (algorithm/hidden). You cannot read or write to these words.

Valid Addresses

The only cells defined as cell type RW are cells 5, 6, and 7.

SN (Serial Number)

The Serial Number (SN) cell type is used for cell 0 only. This is a read-only data word that contains the hardware key's serial number. The value in cell 0 is pre-programmed and cannot be modified. You cannot assign cell type SN to any cell.

Note: Serial numbers range from 0-65535. They are assigned sequentially and can repeat.

Access Code

The SN cell type has an access code of 1 (locked). You can read the serial number but cannot modify it.

Valid Addresses

The only cell defined as type SN is cell 0.

WP (Write Password)

The Write Password (WP) cell type is used for cell 4 only. This cell contains the write password assigned to you by Rainbow Technologies or your distributor. The value is pre-programmed and cannot be modified. You cannot assign cell type WP to any cell.

Your write password gives you the ability to program an undefined (empty) cell or to change a data word cell (access code 0). Changes to any other cell type require the overwrite password.

Access Code

The WP cell type has an access code of 3 (algorithm/hidden). You cannot read or write to this cell type.

Valid Addresses

The only cell defined as type WP is cell 4.

Special Rules for Algorithm Values

The value in the second word of an algorithm descriptor controls two features:

- Whether the algorithm is active or inactive. Only active algorithm descriptors can be used for queries.
- Whether the enhanced algorithm engine is enabled or disabled.

The active/inactive state of an algorithm is controlled by bit 7 of the second (odd-numbered) word of the algorithm descriptor. If this bit is 1, the descriptor is active. If this bit is 0, the descriptor is inactive.

The state of the enhanced algorithm engine is controlled by bit 6 of the second word of the algorithm descriptor. If this bit is 1, the enhanced engine is enabled. If this bit is 0, the enhanced engine is disabled.

Bits 6 and 7 are defined by the first hex character you enter for word 2 of an algorithm descriptor. When you use the Advanced Editor to program your keys, it tells you if the value you entered makes the algorithm active or inactive.

Table 3-4 summarizes the effect of the value of the second word on the algorithm descriptor.

Table 3-4. Defining Word 2 of an Algorithm

	Enhanced Engine Disabled	**Enhanced Engine Enabled**
Algorithm Inactive	0000-3FFF	4000-7FFF
Algorithm Active	8000-BFFF	C000-FFFF

For example, an algorithm with a second word of 1FDC is inactive (disabled) and has the enhanced engine disabled. An algorithm with a second word of D000 is active (enabled for queries) and has the enhanced engine enabled.

Note: For maximum security, use of the enhanced algorithm engine is recommended for all algorithms.

The enhanced algorithm feature is available only with the latest model of the SentinelSuperPro. If you plan to use keys purchased prior to June, 1995, use values from 0000-3FFF (inactive) and 8000-BFFF (active).

Valid Addresses for Algorithms

Certain cell types are designed to be used only in groups. Specifically, algorithm activation passwords (AP) and algorithm counters (CA) are used only in association with algorithm descriptors (cell types AA, AH, and IA). There are address restrictions that apply to the placement of these cell groups in the hardware key. You must be aware of the restrictions when planning your programming layout.

The following combinations of algorithm descriptors, counters, and passwords are supported:

- Algorithm (2 cells)
- Algorithm with password (4 cells)
- Algorithm with counter (3 cells)
- Algorithm with two counters (4 cells)
- Algorithm with password and counter (5 cells)
- Algorithm with password and two counters (6 cells)

If you allow the SentinelWizard to design your key layout, it chooses appropriate locations for your algorithms. If you use the Advanced Editor, you can choose the locations but are not allowed to place an algorithm in an invalid position.

The address restrictions for these cell groups are summarized on the following pages. In this discussion, an algorithm descriptor (shown in the charts as “algo”) can be defined using AA, IA, or AH cells.

Algorithm

An algorithm descriptor that does not have a counter or password can start in any unrestricted cell with an even address.

algo	algo
0 MOD 2	1 MOD 2

Algorithm with password

An algorithm descriptor that has an activation password (AP) must start in a cell with an address equal to 0 MOD 4. The two-word password must immediately follow.

algo	algo	AP	AP
0 MOD 4	1 MOD 4	2 MOD 4	3 MOD 4

Algorithm with one counter

An algorithm descriptor that has one counter (CA) must start in a cell with an address equal to 4 MOD 8. The counter word must immediately precede the algorithm descriptor.

CA	algo	algo
3 MOD 8	4 MOD 8	5 MOD 8

Algorithm with two counters

An algorithm descriptor that has two counters (CA) must start in a cell with an address equal to 4 MOD 8. The counter words must immediately precede the algorithm descriptor.

CA	CA	algo	algo
2 MOD 8	3 MOD 8	4 MOD 8	5 MOD 8

Algorithm with password and one counter

An algorithm descriptor that has both a counter (CA) and an activation password (AP) must start in a cell with an address equal to 4 MOD 8. The counter word must immediately precede the algorithm descriptor, and the two-word password must immediately follow it.

CA	algo	algo	AP	AP
3 MOD 8	4 MOD 8	5 MOD 8	6 MOD 8	7 MOD 8

Algorithm with password and two counters

An algorithm descriptor that has two counters (CA) and an activation password (AP) must start in a cell with an address equal to 4 MOD 8. The counters must immediately precede the algorithm descriptor, and the two-word password must immediately follow it.

CA	CA	algo	algo	AP	AP
2 MOD 8	3 MOD 8	4 MOD 8	5 MOD 8	6 MOD 8	7 MOD 8

Table 3-5 summarizes all valid locations for algorithm descriptors and their associated counters and passwords. For example, the four cells that make up an algorithm with a password can be in locations 08 through 0B, 0C through 0F, 10 through 13, and so on.

When you program an algorithm descriptor using the Advanced Editor, it automatically determines which cell addresses are valid. It does not let you place an algorithm in an invalid location.

Table 3-5. Valid Addresses For Algorithm Words

Valid Addresses for Algorithms

Algorithm Method	**Valid Cell Addresses**
Algorithm (2 cells: algo, algo)	`08-09, 0A-0B, 0C-0D, 0E-0F, 10-11, 12-13, 14-15, 16-17, 18-19, 1A-1B, 1C-1D, 1E-1F, 20-21, 22-23, 24-25, 26-27, 28-29, 2A-2B, 2C-2D, 2E-2F, 30-31, 32-33, 34-35, 36-37, 38-39, 3A-3B, 3C-3D, 3E-3F`
Algorithm with password (4 cells: algo, algo, AP, AP)	`08-0B, 0C-0F, 10-13, 14-17, 18-1B, 1C-1F, 20-23, 24-27, 28-2B, 2C-2F, 30-33, 34-37, 38-3B, 3C-3F`
Algorithm with one counter (3 cells: CA, algo, algo)	`0B-0D, 13-15, 1B-1D, 23-25, 2B-2D, 33-35, 3B-3D`
Algorithm with two counters (4 cells: CA, CA, algo, algo)	`0A-0D, 12-15, 1A-1D, 22-25, 2A-2D, 32-35, 3A-3D`
Algorithm with password and one counter (5 cells: CA, algo, algo, AP, AP)	`0B-0F, 13-17, 1B-1F, 23-27, 2B-2F, 33-37, 3B-3F`
Algorithm with password and two counters (6 cells: CA, CA, algo, algo, AP, AP)	`0A-0F, 12-17, 1A-1F, 22-27, 2A-2F, 32-37, 3A-3F`

Chapter 4 - API Calls

The API functions used to communicate with the SentinelSuperPro driver are summarized in Table 4-1 and described in detail in this chapter.

The API calls are very similar across platforms. The differences are as follows:

- For OS/2, Windows NT, and Windows 95/98, each function name is preceded by *RNBO*. For example, the *sproInitialize()* call is *RNBOsproInitialize()*.
- For OS/2, Windows NT, and Windows 95/98, all functions require a pointer to a packet record (RBP_SPRO_APIPACKET) as a parameter. The SentinelSuperPro driver uses the data in the packet record to communicate with the hardware key. You must allocate memory for the record, and pass its starting address to the API functions. An application program should never modify the data in the packet.

  For DOS and Windows 3.x, all functions require a pointer to a structure called the UNITINFO record. This is used in the same way as the RBP_SPRO_APIPACKET record, but has a different structure.

For suggestions on how different functions can be used to implement various protection strategies, see Chapter 7. For the exact syntax required by your programming language, refer to the sample program in your working directory. For more reference information on the functions, see your language's API.TXT file.

Status codes that may be returned by the API are listed in Appendix A.

Summary of API Functions

Table 4-1 presents a summary of the SentinelSuperPro API Functions:

Table 4-1. Summary Of API Functions

Function	Description
sproActivateAlgorithm()	Activates an inactive algorithm descriptor so that it can be used by the *sproQuery()* function.
sproDecrement()	Decrements a counter word by 1. If the counter is associated with an active algorithm descriptor, decrementing to 0 deactivates the algorithm.
sproExtendedRead()	Reads the value and access code of any unhidden memory word in the key.
sproFindFirstUnit()	Searches all attached keys for a specified developer ID.
sproFindNextUnit()	Searches for the next key with the same developer ID.
sproFormatPacket() *(Win 32 and OS/2 only)*	Validates the size of the packet (RBP_SPRO_APIPACKET) and initializes field defaults.
sproGetVersion()	Returns the SentinelSuperPro driver’s version number.
sproInitialize()	Performs any required initialization of the driver. This function must be called once before any other API function is called.
sproOverwrite()	Changes the value and/or access code of any word except the reserved words in cells 0 through 7.
sproQuery()	Sends a data string to the key, scrambles it using a specified algorithm descriptor, and returns the scrambled string to the application.
sproRead()	Reads the value of any unhidden memory word in the key.
sproWrite()	Changes the value and/or access code of any word with an access code of 0 (read/write data).

Note: For OS/2, Windows NT, and Windows 95/98, each function name is preceded by RNBO.

sproActivateAlgorithm()

The *sproActivateAlgorithm()* function activates a specified algorithm descriptor, making it available for queries. Use of this function requires the write password, as it changes the value in the algorithm descriptor.

To use this feature, you must define an algorithm descriptor that has an activation password associated with it. For details, see Chapter 3.

sproActivateAlgorithm() provides a means for you to prevent an end user from using your application before you give the user a password. Set a required algorithm descriptor to inactive, and write a utility to activate it using the *sproActivateAlgorithm()* function. Then give the activation utility and password to authorized users only. A user without the correct password cannot run the application. For more details on this technique, see **Using Activation Passwords** in Chapter 7.

Parameters

The *sproActivateAlgorithm()* function requires the following parameters:

- A pointer to the packet record or UNITINFO structure.
- Your write password.
- The value programmed into the activation password (the two AP words that follow the algorithm descriptor being activated).
- The address of the first (low-order) word of the algorithm descriptor to be activated. This address must be even. For more information on valid addresses for algorithm words, see Chapter 3.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproDecrement()

The *sproDecrement()* function subtracts 1 from the value in a read/write data word (access code 0) or a counter word (access code 2). Use of this function requires the write password.

If you try to decrement a locked or hidden word, the driver returns **access denied**. If the word already contains the value 0, the driver returns **already zero**. Your application should check for both status codes.

sproDecrement() can be used to limit the number of times a demonstration program can be executed. You associate a counter with an algorithm descriptor, then decrement that counter each time the program is executed. When the counter reaches 0, the algorithm is deactivated automatically. Future queries therefore return invalid responses. For more information on defining an algorithm descriptor with a counter, see Chapter 3.

Parameters

The *sproDecrement()* function requires the following parameters:

- A pointer to the packet record or UNITINFO structure.
- Your write password.
- The address of the counter or data word to be decremented.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproExtendedRead()

The *sproExtendedRead()* function reads the value and access code of any unhidden memory word. An unhidden word has an access code of 0 (read/write data), 1 (read-only data), or 2 (counter).

Algorithm/hidden words, which have an access code of 3, cannot be read. If you try to read a hidden word, the driver returns **access denied**. You can determine that a word is an algorithm word by checking for this status.

Parameters

The *sproExtendedRead()* function requires the following parameters:

- A pointer to the packet record or UNITINFO structure.
- The address to be read.

- A pointer to the contents of the specified word, returned by the driver if the call is successful.
- A pointer to the access code of the specified word, returned by the driver if the call is successful. Possible access codes are **0** (read/write data), **1** (read-only data), and **2** (counter). The call will never return an access code of **3** (algorithm/hidden).

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproFindFirstUnit()

The *sproFindFirstUnit()* function locates a SentinelSuperPro key with a specified developer ID. After calling *sproInitialize()*, you must call this function before using any other calls. If the hardware key is found, the packet or UNITINFO record will contain valid data. If no key is found, the packet or UNITINFO record will be marked invalid.

sproFindFirstUnit() searches all cascaded units connected to any parallel port or USB port. If multiple keys have the same developer ID, *sproFindFirstUnit()* accesses the first key found. If desired, use *sproFindNextUnit()* to find another key with the same developer ID.

Parameters

The *sproFindFirstUnit()* call requires the following parameters:

- • The developer ID assigned to you by Rainbow Technologies or your distributor.
- • A pointer to the packet record or UNITINFO structure.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproFindNextUnit()

The *sproFindNextUnit()* function searches for the next key with a given developer ID. The developer ID is specified using *sproFindFirstUnit()*, which must be called first. (To find a key with a different developer ID, call *sproFindFirstUnit()* again.)

If *sproFindNextUnit()* is successful, the packet or UNITINFO record contains the data for the next hardware key. If not successful, the packet or UNITINFO record is marked invalid. To re-initialize the packet or UNITINFO record, use *sproFindFirstUnit()* and, optionally, *sproFindNextUnit()* again.

sproFindNextUnit() searches all cascaded units connected to any parallel port or USB port. If several keys are attached, your application can call *sproFindNextUnit()* multiple times.

Parameters

The *sproFindNextUnit()* function requires one parameter: a pointer to the packet record or UNITINFO structure.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproFormatPacket() (Win 32 and OS/2 only)

The *sproFormatPacket()* function validates the size of the packet record and initializes its default values. This call must be made before calling any other API functions, or an error 2 (INVALID PACKET) will be returned.

Parameters

The *sproFormatPacket()* requires the following parameters:

- A pointer to the packet record (RBP_SPRO_APIPACKET).
- The value of PacketLen is an integer containing the packet length (1028 bytes).

Return Values

All functions return an unsigned, 16-bit value. A value of zero indicates a successful operation; all other values indicate an error. If an error occurs, the function returns one of the status codes listed in Appendix A. If a non-zero status is returned, then any other data returned by the function will be meaningless.

sproGetVersion()

The *sproGetVersion()* function returns information on the SentinelSuperPro driver being used by the application.

Parameters

The *sproGetVersion()* function requires the following parameters:

- A pointer to the packet record or UNITINFO structure.
- A pointer to the location for the returned major version number.
- A pointer to the location for the returned minor version number.
- A pointer to the location for the returned revision number.
- A pointer to the location for the returned driver type identifier.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproInitialize()

The *sproInitialize()* function lets the driver perform any needed initialization. Your application must call *sproInitialize()* one time before calling any other API function.

Parameters

The *sproInitialize()* function requires no parameters.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproOverwrite()

The *sproOverwrite()* function lets the application change the value and access code of any word except the restricted cells (0 through 7). Use of this function requires both the write and overwrite passwords.

Note: The overwrite password is extremely powerful. Try to restrict its use to your factory instead of putting it in your released application. If your protection scheme does not require the overwrite password from your application, Rainbow can program each of your keys with a “random overwrite password.” Each key will contain a different overwrite key password and can not be changed in the field.

Parameters

The *sproOverwrite()* function requires the following parameters:

- A pointer to the packet record or UNITINFO structure.
- Your write password.
- Your two-word overwrite password.
- The address you want to write to.
- The access code to be assigned to this word: **0** (read/write data), **1** (read-only data), **2** (counter), or **3** (algorithm/hidden).
- The value you want to write to this address.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproQuery()

The *sproQuery()* function lets the application send a query string to the SentinelSuperPro key. The application also specifies the algorithm descriptor to be used to scramble the string.

The key scrambles the input string according to its proprietary algorithm and the information stored in the specified algorithm descriptor. The key then returns the scrambled string to the application. The application compares this response string to the expected result to determine if the correct key is still attached to the parallel port or USB port.

During your development phase, use the Advanced Editor’s **Evaluate API: Query** option to determine the responses your key’s algorithm descriptors will return for given input strings.

Parameters

The *sproQuery()* function requires the following parameters:

- A pointer to the packet record or UNITINFO structure.
- • The number of bytes in the query string (maximum 56).
- A pointer (offset then segment) to the buffer containing the query string.
- A pointer to a buffer where the driver is to return the response string. Make sure this buffer is large enough the hold the entire response string. Alternatively, you can pass a pointer to a location for the driver to return the last 32 bits of the response string.
- The address of the first (low-order) word of the algorithm descriptor to be used for the query. This address must be even. See Chapter 3 for more information on algorithm word addresses.

In general, longer query strings offer greater protection. It is recommended that your query strings be at least eight hex characters (32 bits) long. The maximum length of a query string allowed by the driver is 56 bytes.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproRead()

The *sproRead()* function lets the application read the value of any unhidden memory word. An unhidden word has an access code of 0 (read/write data), 1 (read-only data), or 2 (counter).

Algorithm/hidden words, which have an access code of 3, cannot be read. If you try to read a hidden word, the driver returns **access denied**. You can determine that a word is an algorithm word by checking for this status.

sproRead() returns the word’s value but not its access code. To obtain the access code, use *sproExtendedRead()*.

Parameters

The *sproRead()* function requires the following parameters:

- • A pointer to the packet record or UNITINFO structure.
- • The address to be read.
- • A pointer to the contents of the specified word, returned by the driver if the call is successful.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

sproWrite()

The *sproWrite()* function lets the application write to a word with an access code of 0 (read/write data). Use this function to program or change any data word or undefined (empty) word.

To program a word with an access code of 1 (read-only data), 2 (counter), or 3 (algorithm/hidden), you must use the *sproOverwrite()* function.

Use of the *sproWrite()* function requires the write password.

Parameters

The *sproWrite()* function requires the following parameters:

- • A pointer to the packet record or UNITINFO structure.
- • Your write password.
- • The address you want to write to.
- • The access code to be assigned to this word: **0** (read/write data), **1** (read-only data), **2** (counter), or **3** (algorithm/hidden).
- • The value to be written at the specified address.

Return Values

If successful, the function returns **success**. If an error occurs, the function returns one of the status codes listed in Appendix A.

Chapter 5 - Using the Advanced Editor

The Advanced Editor lets you program your hardware keys directly, rather than letting the Wizard select the locations in which to store your algorithms, data, and counters.

Note: If you use the Advanced Editor, the SentinelWizard cannot generate API pseudo-code for you. You must write the API calls required to use the values programmed into your keys.

For general information on the SentinelWizard, see the *SentinelSuperPro Developer's Guide*.

Entering and Leaving the Advanced Editor

The Advanced Editor is accessed through the SentinelWizard.

While in the SentinelWizard, click the **Go to Advanced Editor** button to launch the Advanced Editor.

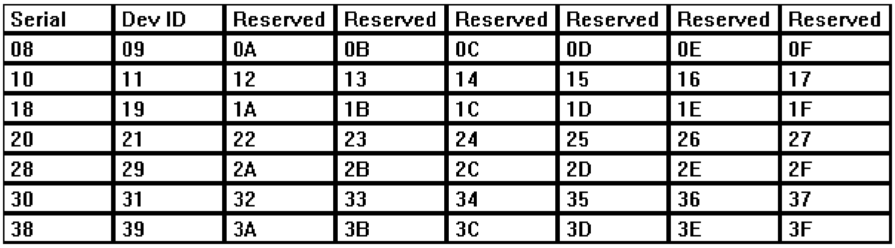
You are prompted for a Profile name. A profile is the set of specifications you define with the SentinelWizard or the Advanced Editor. This profile can be saved and reopened later to program more keys with the same data. <profile name>.DAT is the binary file created when you save your Profile.

Note: For your convenience Rainbow can program your SuperPro keys prior to delivery if you send us your profile .DAT file. Contact your Rainbow representative for more information.

Once in the Advanced Editor, click **Go to Wizard Mode** if you want to return to the guided process.

The Key Matrix

The Advanced Editor displays a matrix of the 64 cells in a hardware key:

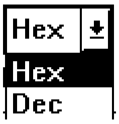


Serial	Dev ID	Reserved	Reserved	Reserved	Reserved	Reserved	Reserved
08	09	0A	0B	0C	0D	0E	0F
10	11	12	13	14	15	16	17
18	19	1A	1B	1C	1D	1E	1F
20	21	22	23	24	25	26	27
28	29	2A	2B	2C	2D	2E	2F
30	31	32	33	34	35	36	37
38	39	3A	3B	3C	3D	3E	3F

The first eight cells are reserved for system information. You can program the other 56 cells.

Choosing Hex or Decimal

You can choose whether you work in hex or decimal. To switch, pull down the **Hex/Dec** box in the upper right corner of the screen and select the option you want.



Your choice affects the way cells are numbered in the matrix and the values you enter.

How Cells Can Be Used

Each cell or group of cells in the key can be programmed in many different ways. The choices are called “item types,” and each cell or group of cells you program is called an “item.” The item types are summarized below.

Use this type...	**If you want to...**
Algorithm (2 cells)	Scramble an input string but do not want an associated counter or activation password.
Algorithm with Counter (3 cells)	Limit the number of times a demo program can be executed.
Algorithm with Counter and Password (5 cells)	Limit the number of times a demo program can be executed, and provide a means for the program to be reactivated in the field.
Algorithm with Password (4 cells)	Require the user to enter a password to get the application to run initially.
Algorithm with Two Counters (4 cells)	Use two counters. The first counter to reach 0 deactivates the algorithm, which usually stops your application from running properly.
Algorithm with Two Counters and Password (6 cells)	Implement an algorithm that is to be deactivated when either of two counters reaches 0. The user must input a password to activate or reactivate the application.
Algorithm Word (1 cell)	Implement an algorithm in which the two words are specified separately. For example, the user or installer may run a program on-site to enter the other half.
Counter Word (1 cell)	Limit the number of times a demo program can be executed, or count the number of times any particular operation is performed.
Data Word (1 cell)	Store a data value your application can read and change during execution.
Locked Data Word (1 cell)	Store a data value your application can read but cannot change without the overwrite password.

Programming Cells

To program a cell in the hardware key, you select the cell and an item type. If the item type requires multiple cells, the cells next to the one you select are allocated automatically.

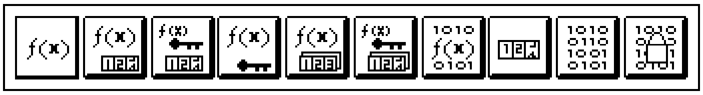
The Advanced Editor does not let you define an item if there is not enough room for it. For example, the Advanced Editor does not let you program an Algorithm with Counter if the cell you choose does not have two empty cells next to it.

To program a cell or group of cells:

1. Move to one of the cells you want to program. Click the right mouse button, select **Create Item**, and select an item type.

Or:

Click the tool bar icon for the item type you want, drag it to an unused cell, and release the mouse button. (To see which item type each icon represents, pass the mouse pointer over it.)



Hint When you drag an icon, the words **Drop Here!** appear when you pass over an appropriate group of cells.

2. Enter the required data as prompted.
3. Click **OK**.

Changing Cells

Once cells are defined, you can change the stored values — for example, the value in a data word or an algorithm descriptor.

You cannot change the item type assigned to the cell. Instead, you must delete the item, then add it back using the new item type.

To change the value in a cell:

1. Click the right mouse button on the item you want to change.
2. Select **Edit Item**.
3. Modify the cell value(s) as desired.

4. Click **OK**.

Clearing Cells

To delete an item, right-click on it and select **Delete Item**. All cells used for that item will be cleared.

To delete an item:

1. Click the right mouse button on the item you want to delete.
2. Select **Delete Item**.
3. To confirm, click **Delete Item**. To keep the item as is, click **Cancel**.

Updating the Attached Key

The **Update Key** button at the top of the window indicates if the attached key and the matrix on the screen are in sync.

The button is enabled until the key is programmed during the session or when the matrix is changed after the key is updated. If you have used the Wizard to program your key, the **Update Button** will be disabled unless you make a change in the matrix.

To update all cells in the attached key:

1. Click the **Update Key** button. If this button is not enabled, the key’s memory has already been updated to match the matrix.
2. Click **Yes** to confirm the write, or **No** to cancel.
3. Click **OK**.

Querying the Attached Key

When your application sends a query to an algorithm descriptor in your key, it compares the scrambled response to the response it expects. To determine the expected responses, you must query the algorithm during your development phase. You can do this in the Advanced Editor by selecting an algorithm item and then selecting **Evaluate API: Query**.

To send a query to the attached key:

1. If the **Update Key** button is enabled, the data on your screen does not match that programmed into the attached key. Click **Update Key** to update the cells in the key.
2. In the matrix, move to the item for the algorithm you want to query. Watch the hint text that appears below the selected item to make sure you have the one you want.
3. Click the right mouse button.
4. Select **Evaluate API**.
5. Select **Query**.
6. Type the query string.
7. Click **OK**.
8. The Advanced Editor displays the result of the query. The value next to **Response** is the scrambled string your application should expect to receive when it sends the same query.

Using Other API Calls

Right-clicking on an item and then selecting **Evaluate API** displays a menu listing API calls. Select an API call to have it performed. For example, select **Read** to read the selected cell, or **Write** to write data to it.

For detailed information on each API call, see Chapter 4.

Selecting an API call may lead to a dialogue box where you enter required information. If you select **Write**, for example, you are prompted for the data to be written.

The API call is then made, and a box appears with the result from the driver. A successful result is indicated by status code 0000. For help on unsuccessful responses, see Appendix A.

The result box also shows the details of the API call, such as the cell address and any data specified.

```
Input Parameters:
 address = 21
 data = 3333
 accessCode = 0
Function Result = Success 0000

*****************************************************
*
 C Language Prototype:

 int sproOverwrite(UNITINFO   *unitinfo,
          unsigned int  writePassword,
          unsigned int  overwritePassword1,
          unsigned int  overwritePassword2,
          int      address,
          unsigned int  data,
          unsigned char accessCode);
```

Chapter 6 - Using the Driver Encryption Tools

This chapter applies only to developers of DOS applications and Windows 3.x applications using a linked-in (non-DLL) driver.

The SentinelSuperPro Driver Encryption Toolset provides you with an extra level of protection from potential hackers for DOS and Windows 3.1 applications using the linked-in driver. Using the Driver Encryption Tools, you can substantially enhance the level of security by hiding the code that communicates with the hardware key.

The Driver Encryption toolset provides the following:

- A choice of four encryption methods.
- The option of specifying the starting encryption seed value.
- The option of specifying a modifier to be applied to the encryption seed value during the encryption process. The modifier is applied after each byte in the driver is encrypted.
- The ability to encrypt the entire driver or a specified range of bytes. Different encryption methods can be applied to different portions of the code, as long as each portion is decrypted appropriately at runtime.
- The ability to re-encrypt the driver after runtime decryption.
- The ability to calculate checksums to validate the driver file before and after encryption/decryption.

Using the toolset, you encrypt the SentinelSuperPro driver object code file based on parameters you specify. You link your application to the encrypted driver file, then call routines from within your application to decrypt the driver prior to communicating with the hardware key. This process allows your application to have an “invisible” copy of the driver on disk and in memory when the driver code is not in use.

Each language interface directory contains a file that defines the routines or macros you will use to decrypt and optionally re-encrypt the driver from your protected application. The file extension varies by language; the file name is usually SPROCRYP.

Note: Driver encryption is not supported for OS/2, Windows NT, Windows 95/98 or Windows with dynamic (DLL) linking.

Using Driver Encryption with Windows Applications

If you have a statically-linked Windows application and allow multiple instances to run simultaneously, attempting to encrypt or decrypt the driver may cause a General Protection Fault. This occurs because all instances of the application share the same code segment. There are two ways to deal with this situation:

- Do not change the values of the encryption or decryption parameters at runtime. This ensures that any instance of the application will encrypt or decrypt the driver the same way every time. You must also encapsulate the entire process (decrypt, API operation, encrypt) in a critical section so that any instance of your application does not try to decrypt code that is already decrypted.
- Do not use the encryption/decryption feature when running under Windows.

The Driver Encryption Process

SPROX.EXE, the SentinelSuperPro driver encryption tool, is used to encrypt the driver before the link process. After linking the encrypted driver to your protected application, the resulting executable file contains encrypted code that must be decrypted at runtime, before accessing the SentinelSuperPro driver.

The overall driver encryption/decryption process is as follows:

1. You run SPROX to encrypt the SentinelSuperPro driver. SPROX can be run interactively or all parameters can be specified on the command line.
2. You modify your protected application to decrypt the driver (using the same parameters you specified with SPROX) so you can communicate with the hardware key.

3. You link your protected application with the encrypted driver file.

The driver is therefore shipped in an encrypted form, and is decrypted only at runtime.

Note: For added security, the SentinelSuperPro driver is released in an unusable (unlinkable) state. You must run SPROX against the driver file to produce a legal object code file.

For convenience, SPROX provides an option that generates a legal object code file without applying encryption. Therefore, you do not have to add decryption code to your protected application. You can use this option during the development phase, but should not use it for your final product.

You can run SPROX more than once on the same file, producing layers of encryption. At runtime, however, the driver must be completely decrypted before it can be accessed. Layered encryption requires decryption to occur in reverse order; that is, the last area encrypted with SPROX must be the first decrypted at runtime.

SPROX error messages are described in Appendix B. If you receive an error while running SPROX interactively, press any key except ESC to continue. If you want to terminate SPROX, press ESC. Some errors cause SPROX to abort. Any error generated when SPROX is run with command-line parameters causes execution to abort.

Available Encryption Methods

Four methods can be used to encrypt the driver file: XOR, ADD, ROTATE, and SWAP. These methods are described in Table 6-1.

For each method, you provide the initial encryption seed value and a modifier to be applied to the seed value after each byte is encrypted.

Table 6-1. Encryption Methods

Method	Description
XOR	Performs an exclusive OR against each byte in the SentinelSuperPro driver using the current encryption seed value.
ADD	Adds the current encryption seed value to each byte in the SentinelSuperPro driver.
ROTATE	Rotates the bits in each byte in the SentinelSuperPro driver left one or right one, depending on the current encryption seed value.
SWAP	Swaps all byte pairs in the SentinelSuperPro driver. At the current time, this method does not use the encryption seed value or modifier. This method requires an even number of bytes in the encrypt/decrypt range.

Running SPROX from the Command Line

When you execute SPROX, you can enter parameters on the command line to specify all desired encryption options. The full syntax for executing SPROX is:

```
SPROX.EXE [{-,/}<parameter> . . . {-,/}<parameter>]
```

Each parameter is a one-character code followed immediately by the value for that parameter. Each parameter must be preceded by a space and a dash or a slash. Parameters can be entered in any order, in upper- or lowercase.

After you execute SPROX using valid command-line parameters, it displays a summary screen that shows:

- The name of the encrypted file.
- The name of the decryption routine you must call from your protected application to decrypt the file.
- The values of the parameters you must pass to the decryption routine.
- The before and after checksums calculated for the SentinelSuperPro driver.

If desired, you can use a command-line parameter to have the above information written to a log file instead of the screen.

Summary of Parameters

Table 6-2 summarizes the parameters that can be specified on the SPROX command line. Note the following:

- If no parameters are specified on the command line, SPROX prompts you for the needed information. SPROX is also invoked interactively if you specify only **/V**, **/Q**, **/L**, and/or **/U**.
- You must specify **/I** if you use **/O**, **/M**, **/S**, **/A**, **/B**, or **/N**.
- The values listed in the **Default** column in Table 6-2 are applied if you specify **/I** but omit the specified parameter. Note that some of the defaults provided are randomly generated. You may wish to provide your own values (that is, do not accept the defaults) so you can run SPROX via a make file.

Table 6-2. SPROX Command-Line Parameters

Parameter	Description	Default
/A{0...255}	The value by which the encryption seed value is modified after each byte is encrypted.	random
/B{0..end of code}	The offset (inclusive) in the driver file at which to begin encrypting. It is recommended that you encrypt the entire file (/B0).	0
/H or /?	Provides help on all SPROX parameters.	
/I<filename>	The name of the input file (the driver file to be encrypted).	
/L<filename>	Logs SPROX summary information to the specified file instead of the screen. Creates the file if it does not exist; appends to it if it already exists. Each summary is date/time-stamped.	standard out
/M{X,A,R,S}	The encryption method to apply: X (XOR), A (ADD), R (ROTATE), or S (SWAP). See Available Encryption Methods for details.	random
/N{0..bytes in code}	The number of bytes in the driver file to encrypt. 0 represents the last byte in the file (inclusive); use 0 instead of the actual byte count to avoid having to modify your application if the driver size changes. It is recommended that you encrypt the entire file (/N0).	0
/O<filename>	The name of the output file (the encrypted driver file).	SPRO!!!!.OBJ
/Q	Invokes quiet mode: errors are displayed but sign-on/sign-off messages are suppressed.	
/S{1..255}	The initial encryption seed value.	random
/U	Produces an unencrypted (but legal) output file. This method is not recommended.	
/V	Overwrites the output file if it already exists. If you are using command-line parameters, you do not specify /V, and the file named by /O already exists, SPROX aborts. If you are running interactively, you are asked if you want to overwrite the file.	

Command-Line Examples

The following examples demonstrate the effect of various command-line parameters on SPROX.

`SPROX /Isuperpro.obj`
Encrypts the SUPERPRO.OBJ file, creating an output file called SPRO!!!!.OBJ (the default). SPROX aborts if SPRO!!!!.OBJ already exists. The encryption method, initial seed value, and seed modifier are *assigned randomly* by SPROX — different values are generated each time. Because no byte range was specified, the entire driver is encrypted. This is the minimum acceptable command line.

`SPROX /Isuperpro.obj /V /Mx /S52`
Encrypts SUPERPRO.OBJ using the XOR encryption method, creating an output file called SPRO!!!!.OBJ (the default). The output file is overwritten if it already exists. The initial encryption seed value is 52. The seed modifier is assigned randomly. Because no byte range was specified, the entire driver is encrypted.

`SPROX /Lencrpyt.log`
Invokes SPROX interactively because **/I** was not specified. SPROX will record the summary information (details on the encryption parameters specified or selected by random) in a file called ENCRYPT.LOG.

`SPROX -ix.obj -oxt2.obj -mA -s2 -a25 -n1400`
Encrypts the X.OBJ file using the ADD encryption method, creating an output file called X2.OBJ. SPROX aborts if X2.OBJ already exists. The initial seed value is 2 and the seed modifier is 25. Bytes 0 through 1399 of the SentinelSuperPro driver (for a total of 1400 bytes) are encrypted.

Running SPROX Interactively

If you execute SPROX with no parameters on the command line, the utility prompts you interactively for all required encryption specifications.

SPROX is also invoked interactively if you enter no command-line parameters other than **/L**, **/V**, **/Q**, and **/U**. You may wish to use **/L** on the command line to write the encryption parameters to a log file. (By default, SPROX displays the parameters and checksum information on the screen.)

SPROX shows the default value, if one is provided, in parentheses after each prompt. To accept the displayed default, press ENTER without typing an entry.

To run SPROX interactively:

1. Move to the directory in which SPROX is installed.
2. Type **SPROX** and press ENTER.
3. Read the introductory screen, then press ENTER.

```
File to encrypt =>
```

4. Enter the name of the SentinelSuperPro driver object file (usually SUPERPRO.OBJ). Press ENTER.

```
Output file (Press ENTER for SPRO!!!!.OBJ) =>
```

5. Enter a name for the output (encrypted) driver file and press ENTER. If you want the file to be named SPRO!!!!.OBJ, just press ENTER.

If the file already exists, you are asked if you want to overwrite it. Type **Y** to replace the existing file, or **N** to specify a different name for the new file.

```
Encryption method: Xor, Add, Rotate, Swap
  (Press ENTER for a random selection) =>
```

6. Type the first letter (**X**, **A**, **R**, or **S**) of the encryption method you want and press ENTER. If you want SPROX to select a method randomly, just press ENTER. See **Available Encryption Methods** for descriptions.

```
Starting encryption seed: 1..255
  (Press ENTER for a random selection) =>
```

7. Type the encryption seed value you want (**1** through **255**) and press ENTER. If you want SPROX to select a value randomly, just press ENTER.

```
Seed modifier: 0..255
  (Press ENTER for a random selection) =>
```

8. Type the seed modifier you want (**0** through **255**) and press ENTER. If you want SPROX to select a modifier randomly, just press ENTER.

```
Encryption Range: <1st byte: 0..1937><space><Number of bytes:
0..1938>
  (Press ENTER to specify the entire file) =>
```

9. If you want to encrypt the entire file, press ENTER. If you want to specify a byte range, type the following and press ENTER:

- The first byte to be encrypted (zero-relative). Your entry must be between 0 and the number displayed (the last byte in the file).
- A space.
- The total number of bytes to be encrypted. Your entry must be between 0 and the number displayed (the file's total byte count). Enter **0** if you want to encrypt through the last byte of the file. If you selected SWAP encryption, this number must be even.

```
File to encrypt => SUPERPRO.OBJ
Output file           => SPRO!!!!.OBJ
Encryption method     => X
Encryption Seed => 52
Seed Modifier   => 162
Encryption range      => 0..1937 (1938 bytes)
Is this information correct (Y/n)?
```

10. Review the displayed data. If it is correct, type **Y** and press ENTER. If not, type **N**, press ENTER, and go back to step 4.

```
The resulting encrypted SuperPro Object file is => SPRO!!!!.OBJ
```

11. If you did not enter **/L** on the command line, a summary screen appears. Make a note of the following important information:
    - The name of the encrypted file.
    - The name of the decryption routine you must call from your protected application to decrypt the file.
    - The values of the parameters you must pass to the decryption routine.
    - The before and after checksums calculated for the driver file.

Encrypting and Decrypting from Your Application

After encrypting the SentinelSuperPro driver with SPROX, you must add code to your protected application to decrypt the driver prior to sending commands to the hardware key. If desired, you can also re-encrypt the driver from within your protected application.

If you ran SPROX multiple times to generate layers of encryption, your protected application must decrypt the driver the same number of times, in reverse order. For example, if you encrypted the file with XOR, then with ADD, you must decrypt with ADD, then with XOR.

Routines or macros are provided to perform both decryption and encryption, using any of the four available methods. A routine is also provided for calculating the driver code checksum.

The information on the following pages has been generalized. Refer to the interface file for your development language for specific interface details.

Encryption and Decryption Routines

Routines or macros are provided to support the four available encryption methods (XOR, ADD, ROTATE, and SWAP). Usually, six routines are provided, although some language interfaces have eight. The routines are summarized in Table 6-3.

All encryption/decryption routines require the following parameters:

- The initial encryption seed value.
- The seed modifier to be applied after each byte is encrypted.
- The first byte in the file to encrypt/decrypt.
- The number of bytes to be encrypted/decrypted.

Note: The decryption/re-encryption routines do not verify byte ranges at runtime. Be careful not to access areas outside the SentinelSuperPro driver.

Table 6-3. Encryption/Decryption Macros

Encryption Method	Routine/ Macro	Description
XOR	CryptX	Encrypts by XOR’ing the seed against each byte.
	DeCryptX	Decrypts by XOR’ing the seed against each byte. This routine may be the same as CryptX.
ADD	CryptA	Encrypts by adding the seed to each byte.
	DecryptA	Decrypts by subtracting the seed from each byte.
ROTATE	CryptR	Encrypts by rotating the bits in each byte by 1 left or right, depending on the current seed value.
	DecryptR	Decrypts by rotating the bits in each byte back to the original position.
SWAP	CryptS	Encrypts by swapping byte pairs. This method requires an even number of bytes.
	DeCryptS	Decrypts by swapping byte pairs. This method requires an even number of bytes. This routine may be the same as CryptS.

Checksum Routine

When you run SPROX to encrypt the driver object file, it calculates before and after checksums. After decrypting the driver from your protected application, you may wish to verify the checksum to ensure that the object file is intact.

One routine or macro is provided to calculate the checksum of a specified byte range in the SentinelSuperPro driver file. This routine, ChkSum, generates a long checksum.

The parameters required by ChkSum are:

- The first byte in the file to include in the checksum calculation.
- The last byte in the file to include in the checksum calculation.
- The variable to contain the result of the checksum. Note that for most languages, the ChkSum macro does not zero this variable first.

Remember that SPROX reports the before and after checksum calculations each time it is executed.

Chapter 7 - Advanced Protection Techniques

The goal of any software protection scheme based on the SentinelSuperPro is to significantly reduce the chance that someone can defeat the protection and use your software without the hardware key. In general, the time and expense required for a skilled pirate to break your scheme is directly related to the number and complexity of the locks you place in your software. Protection can be as simple or as complex as you wish.

This chapter describes several methods you can use, singly or together, to protect your application. Many of these schemes are based on one or more of the following general methods:

- **Overload** potential pirates with data by calling the hardware key many times throughout your code.
- **Decentralize** your locks throughout the code, rather than restricting them to a few places that can be easily detected and eliminated.
- **Distract** potential pirates with locks that make your application perform long series of meaningless operations. These calls mislead hackers and make your valid locks harder to isolate.

Some techniques can be used with returned values sent from any of the three types of words (data, counter, and algorithm descriptors). Other techniques can be used with only one type of word. The term “returned value” is used for the response from any type of cell.

Once you have an idea of the types of protection strategies you want to use, you must consider how you want to program the 56 non-restricted cells of the hardware key to implement your strategies.

A primary consideration is the number of applications that will share the key. Any of your applications can use any cell, but you must make sure to allow enough cells for the type of protection you want for each application.

After you have planned your configuration design, use the SentinelWizard’s Advanced Editor to program the hardware keys with your chosen values.

About the Examples

All values and cell numbers used in the examples in this chapter are in hexadecimal format.

For simplicity, standard error-checking steps are omitted from the examples. If you receive an invalid response to a query or another function, *always retry the operation before taking action*.

As explained in Chapter 6, you must encrypt linked-in drivers with the SPROX utility before linking them to your application. Within your code, you must call a routine to decrypt the driver before each API call, then optionally call another routine to re-encrypt the driver afterward. For simplicity, these routines are not included in the examples.

The API function names referenced in the examples are those used for DOS and Windows. For OS/2, Windows NT, and Windows 95/98, each function name must be preceded by *RNBO*.

Basic Guidelines

To ensure that your protection scheme is effective, follow these basic guidelines:

- **Send queries frequently.** One of the most basic and effective techniques you can use to confound hackers is to call the SentinelSuperPro key frequently. If you rely on a single call at the beginning, it is relatively easy for a skilled pirate to isolate the call and defeat your protection.

Another potential problem with querying only once is that a user could remove the key after starting the application. The key could then be used to run another copy of the application. The first copy would continue to run since no queries are being performed to check for the key’s continued presence.

- **Scatter lock components.** Software locks consist of multiple steps: calling the key, evaluating the returned value, and acting on the evaluation results. For added protection against piracy, separate these lock components in your code. A software lock is harder to break if its components are physically separated into different sections of the application than if they are located together.

- **Manipulate data.** Use the data returned from the SentinelSuperPro in various ways. For example, leave the result in a global variable, then check it later.

Reading Stored Data

For a simple protection scheme, simply read a cell in the key and verify that it contains the correct data. If it does, continue execution. If the correct data is not found, assume the key is not attached and proceed accordingly.

Example

In this example, you program one cell in the hardware key with a two-byte value. You then have your application read that cell during execution, taking appropriate action after the read.

1. Select a two-byte value. We will use **1234**.
2. Choose a cell in which to program this value. We will use **cell 20**.
3. Use the Advanced Editor to program **cell 20** with the value **1234**. Use the item type **Data Word** if you want the application to be able to modify the cell later. Use **Locked Data Word** if you want the cell to be read-only.
4. Embed the required API functions in your application code. Code your application to display an error or abort if the read operation does not return the value you programmed. (Of course, you should retry the operation before taking a negative action.)

To meet these objectives, your code must contain the following API calls:

- *sproInitialize()* - Performs required initialization.
- *sproFindFirstUnit()* - Establishes communication with the key and updates the packet record or UNITINFO structure.
- *sproRead()* **-** Reads the cell and returns the value in it.

Using Algorithms to Scramble Data

For more protection, you can send a data string to the key, have it scrambled using a pre-programmed algorithm descriptor, then examine the returned value. You can simply verify that the correct scrambled string was returned, or use that value to control your program's execution in some way.

Remember that longer query strings generally offer greater protection. It is recommended that your query strings be at least 32 bits (eight hex characters) long.

Example

This example demonstrates how to set up your application to require a correctly scrambled response from the hardware key.

1. Select two 16-bit hex values to use for the algorithm descriptor. We will use **1234** and **C000**. Remember that the second word must between 8000 and FFFF to make the algorithm active.
2. Choose two cells in which to program these values. We will use **cells 0A** and **0B**. Remember that the first address must be even.
3. Choose an input string (preferably at least 32 bits long) to send to the key to be scrambled. We will use **8FA31B4B**.
4. Using the Advanced Editor, create an **Algorithm** item at location **0A**. Enter **1234** for the first word and **C000** for the second.
5. Use the **Program** command to program a test key with your specifications.
6. Right-click on cell **0A**, select **Evaluate API**, then select **Query**. Determine the scrambled value your key will return for a query string of **8FA31B4B**. We will assume the returned value is **1BC235B1**.
7. Embed the appropriate API calls into your application to make the query. Code your application to display a message and exit if the query does not return the correct value (1BC235B1).

To meet these objectives, your code must contain the following API calls:

- *sproInitialize()* - Performs required initialization.
- *sproFindFirstUnit()* - Establishes communication with the key and updates the packet record or UNITINFO structure.
- *sproQuery()* **-** Sends the query string and points to a location for the response string.

Using a Scrambled Value to Encrypt Code

You can use a scrambled value from the hardware key to encrypt and decrypt a portion of your application.

During development, you query the key and obtain a scrambled string, and then use this string to encrypt your application code. You then design your application to query the key again during execution. If the key is present, it returns the string needed to decrypt your code.

Example

1. Select two 16-bit hex values to use for your algorithm descriptor. We will use **4D59** and **F123**. Remember that the second word must be between 8000 and FFFF to make the algorithm descriptor active.
2. Choose two cells in which to program these values. We will use **cells 0A** and **0B**. Remember that the first address must be even.
3. Choose an input string (preferably at least 32 bits long) to send to the key to be scrambled. We will use **7009AB12**.
4. Use the Advanced Editor to assign an **Algorithm** item at address **0A**. Enter **4D59** for the first word and **F123** for the second.
5. Using the **Program** option, program a test key with your specifications.
6. Right-click on cell **0A**, select **Evaluate API**, then select **Query**. Determine the scrambled value your test key will return for a query string of **7009AB12**.

We will assume the returned value is **289CA110**. This will be the encryption seed with which you will encrypt part of your code.

7. Select an encryption method. We will use the Boolean operator XOR.
8. Select the essential data in your code that you want to encrypt. We will use the hex value **8FA31B4B**.
9. Apply the selected operator to the data, using the encryption seed. For this example, the result is **A73FBA5B**. (See **Encryption Techniques** in this chapter for details on the XOR operator.)

If you ship your application with the encrypted code, it will not execute correctly until the code is decrypted by a query.

10. Design your application so that it decrypts the encrypted code if the hardware key is present.

To do this, query the hardware key. If the key is present, the *sproQuery()* function will return the response string of **289CA110** that you used for an encryption seed.

Because XOR is a reversible operation, applying the same encryption seed to the encrypted data will return the data to its original state, and your program should continue to execute properly.

To meet these objectives, your code must contain the following API calls:

- *sproInitialize()* - Performs required initialization.
- *sproFindFirstUnit()* - Establishes communication with the key and updates the packet record or UNITINFO structure.
- *sproQuery()* - Sends the query string and points to a location for the response string.

Using Returned Values as Variables

Because software is generally easier to break than hardware, most pirates will try to break your package by attacking the software. Therefore, any tricks or traps you can implement in your code by incorporating a response from the hardware key will add even more protection.

One effective technique is to disguise software locks in a high-level language by using SentinelSuperPro values to control program flow. With this method, a value returned by the key becomes a logical pointer or selection key to the next execution step or the next subroutine. This makes analysis of your code more difficult.

Another way to use a returned value is to add it to the value of a variable so that the sum is the desired value of the variable. If the variable is used in other parts of the code, then that code is dependent on the call to the SentinelSuperPro.

Example

Suppose that at some point in your application you want a variable to contain the value 13.

Assume that one of the query strings you send to the SentinelSuperPro returns the decimal number 12,345. Set the variable to -12,332, send the query, and add the response to the variable. If the proper key is attached, the variable will contain the correct value.

Controlling Demo Applications

You can use a counter word to limit the number of times an application (usually a demo version) can be executed. You set the counter to the desired limit, then subtract 1 each time the program is run. The decrement is performed using the *sproDecrement()* function.

If desired, you can simply check the counter to see if it has reached 0, then proceed accordingly. Using this method, the word you decrement and read can be either a data word or a counter.

For more secure protection, you can tie the counter to an algorithm descriptor that the application requires for queries. When the counter reaches 0, the associated algorithm is deactivated automatically. (*sproDecrement()* changes the high-order bit of the algorithm descriptor’s second word.) Future queries that use this algorithm return incorrect responses.

To limit program executions using an algorithm/counter, you must program the key to meet the following specifications:

- The word you decrement must be a counter word.
- The counter must be located at an address equal to 3 MOD 8.
- The two words immediately following the counter (at addresses equal to 4 MOD 8 and 5 MOD 8) must contain an active algorithm descriptor.

Note: The relationship between a counter word (in an address equal to 3 MOD 8) and an adjacent algorithm descriptor exists even if you do not intentionally plan it. The algorithm will be deactivated when the counter reaches 0.

You can also use two counters: the algorithm is deactivated when *either* counter reaches 0. The second counter must be located at an address equal to 2 MOD 8.

If you want to be able to reactivate the application after it has been shut down, you must define an activation password. This is a two-word value immediately following the algorithm descriptor. See **Using Activation Passwords** in this chapter for details.

Remember that the counter will still be 0 after the algorithm is reactivated, so make sure your application checks for the **already zero** status from the driver. (You could reset the counter and do decrements again, but this would require putting your overwrite password into the activation utility. Generally, you should avoid using your overwrite password in the field.)

Any of the following item types can be used to program an algorithm with a counter: Algorithm with Counter, Algorithm with Counter and Password, Algorithm with Two Counters, or Algorithm with Two Counters and Password.

The following key layout illustrates an algorithm with one counter and a password:

CA	AA	AA	AP	AP
0B	0C	0D	0E	0F

Cell 0B contains the counter you are decrementing. Cells 0C and 0D contain the active algorithm descriptor (cell type AA). The second AA word must be between 8000 and FFFF—this value will be changed automatically when the counter reaches 0.

Cells 0E and 0F contain the activation password. This is the password that is required to reactivate the application after the counter reaches 0.

Example

This example demonstrates how to limit the number of times a demonstration application can execute. This is done by requiring the application to query an algorithm descriptor that becomes unusable (deactivated) after a specified number of executions. The algorithm descriptor is associated with a counter that is initialized to the number of times the program can be run.

Remember that if you want to let the user reactivate the application after the counter has reached 0, you must also program an activation password.

1. Select the cells you want to use for the algorithm and counter. (Review the address restrictions in Chapter 3.) We will use the following cells:

counter	algo 1	algo 2
0B	0C	0D

2. Decide how many times you want the demo program to run. We will use **5**.
3. Select two 16-bit hex values to use for the algorithm descriptor. We will use **1234** and **C000**. Remember that the second word must be between 8000 and FFFF to make the algorithm descriptor active.

4. Select an input string (preferably at least 32 bits long) to send to the key. We will use **ABCDDCBA**.
5. Use the Advanced Editor to program an **Algorithm with Counter** item at address **0B**:

Algorithm:	1234	C000
Counter:		5

Note that if you want to provide the ability to reactivate the algorithm in the field, you must use the **Algorithm with Counter and Password** item type instead. This item type lets you program an activation password into the two cells following the algorithm descriptor.

6. Add the appropriate API calls to your application:
   - *sproInitialize()* - Performs required initialization.
   - *sproFindFirstUnit()* - Establishes communication with the key and updates the packet record or UNITINFO structure.
   - *sproDecrement()* - Decrements the counter by 1. Call this function every time your application executes. On the fifth execution, the counter in cell 0B reaches 0. At this point, *sproDecrement()* deactivates the algorithm descriptor used by the *sproQuery()* function (see below). On the sixth execution, therefore, the query does not return the expected value. Usually, you would code the application to display a message and then terminate.
   - *sproQuery()* - Sends the query string and specifies a location for the response string.

To make a potential hacker’s task more difficult, separate the *sproQuery()* and *sproDecrement()* function calls in your code. This helps obscure the connection between them.

Using Activation Passwords

You can program the hardware key so that an algorithm descriptor is protected by an activation password. The algorithm descriptor and password are each two words long. The algorithm password must immediately follow the algorithm descriptor. (See Chapter 3 for details on address restrictions.)

For example, cells 14 through 17 can be used as follows:

algo desc 1	algo desc 2	algo password 1	algo password 2
14	15	16	17

By defining an algorithm descriptor with an activation password, you can prevent a user from using your application before you have supplied a password.

The basic steps to this process are the following:

1. At the factory, set the algorithm descriptor to inactive.
2. Design your application so that it executes only after receiving a response string from the (currently inactive) algorithm descriptor.
3. Write a utility that uses the *sproActivateAlgorithm()* API function to activate the algorithm, once the user provides a password in the field.
4. After buying your application, the user runs your initialization utility, entering the password you provide. The algorithm will now return the correct query response string, thereby allowing the protected application to execute.

For added security, you may wish to use a different activation password for each customer (each key/utility).

Example

This example demonstrates how to activate a disabled application. Usually, an application is deactivated because a demo application has been “turned off” after the specified number of executions. Alternatively, you may have set up your application so that the user must enter an activation password before the application will run.

You temporarily “turn off” an application by including a *sproQuery()* call that requires the hardware key to return a correctly scrambled string. Then, you make it impossible for the hardware key to return the string because its algorithm descriptor has been set to inactive.

By definition, an algorithm descriptor is inactive if the high-order bit of its second word is 0. This is done as follows:

- If a counter is used, the *sproDecrement()* function sets the bit to 0.
- In the factory, use the Advanced Editor to set the second word of the algorithm descriptor to a value between 0000 and 7FFF.

You must also write a utility that activates the algorithm descriptor, once the user provides the correct password. The query performed by the protected application will then return the correct response, and the application will run successfully.

The following example assumes that you release your application in a deactivated state, and provide a password and utility to activate it.

1. Use the Advanced Editor to program an **Algorithm with Password** at address **0C**, as follows:

Algorithm:	0123	3456	
Password:		AB16	09C5

Note that the value in the second word of the algorithm descriptor must be between 0000 and 7FFF to make the algorithm inactive.

2. Write a utility with which the user can enter the password you supply.
3. Embed the required API calls in your application to query the hardware key using the activated algorithm descriptor.

To meet these objectives, your code must contain the following API calls. Note that this describes the activation utility, not the protected application that is being activated.

- *sproInitialize()* - Performs required initialization.
- *sproFindFirstUnit()* - Establishes communication with the key and updates the packet record or UNITINFO structure.
- *sproActivateAlgorithm()* - Passes the string input by the user, your write password, and the address of the first word of the algorithm descriptor. If the password is correct, *sproActivateAlgorithm()* changes the algorithm descriptor's active/inactive bit to active, making it available for queries.

You may wish to send a query using the algorithm descriptor before calling *sproActivateAlgorithm()*. If the query returns the correct response, the algorithm is already activated.

Querying Activation Passwords

Normally, an algorithm password is used to activate an inactive algorithm descriptor, as described in **Using Activation Passwords**.

Note that an algorithm password has an access code of 3, meaning that it is an algorithm word. You can therefore use the password itself as an algorithm descriptor.

To make a password an active algorithm descriptor, you must set bit 7 of its second word to 1. (Using the Advanced Editor, set the value in the second word to a number between 8000 and FFFF.) You can then use the *sproQuery()* API function to send an input string to the key, specifying the starting address of the password. The key scrambles the input string according to the bit pattern of the algorithm password, and the driver returns the scrambled response string to your application.

This technique provides an alternate method of querying the key. For example, you may want to query the algorithm password before invoking the *sproActivateAlgorithm()* function, to verify that the password appears to be correct. Note that you cannot do this if you use a different activation password for each customer.

Using One Key for Multiple Applications

One SentinelSuperPro key can be used to protect multiple applications. Usually, you do this by designating certain cells for each application.

The sample layout below illustrates how you can use one key to protect seven applications. Each application is assigned a group of cells consisting of two data words (cell type DW), two algorithm counters (cell type CA), one inactive algorithm descriptor (cell type IA), and an activation password (cell type AP).

	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
00	SN	DI	OP	OP	WP	RW	RW	RW
08	DW	DW	CA	CA	IA	IA	AP	AP
10	DW	DW	CA	CA	IA	IA	AP	AP
18	DW	DW	CA	CA	IA	IA	AP	AP
20	DW	DW	CA	CA	IA	IA	AP	AP
28	DW	DW	CA	CA	IA	IA	AP	AP
30	DW	DW	CA	CA	IA	IA	AP	AP
38	DW	DW	CA	CA	IA	IA	AP	AP

Application 1 would use cells 8 through F, application 2 would use cells 10 through 17, and so on. The values in the restricted cells (0 through 7) are shared by all of the applications.

Obstructing Debuggers

Many potential pirates use debuggers to break large, complex software packages with high licensing fees. You may want to incorporate safeguards aimed directly at preventing the use of debug programs to circumvent software locks. For example, you might lock out the keyboard during SentinelSuperPro queries, or destroy the contents of interrupt vectors 1 and 3 (the trace and breakpoint interrupts).

While no technique can deter every pirate, the more safeguards you use, and the greater the variety you use, the more difficult the pirate’s task. Eventually it makes more sense for potential pirates to either purchase your product or pirate a different, less secure package.

Assembler Language Techniques

Implementing SentinelSuperPro protection in assembly language offers more flexibility than other languages. Note, however, that you can use only one SentinelSuperPro interface subroutine to make hardware key queries. If you try to link two different interface subroutines with your application, you may get doubly defined symbols.

Many potential pirates use debug programs to break large, complex software packages with high licensing fees. You may want to incorporate safeguards aimed directly at preventing the use of debug programs to circumvent software locks.

While no technique can deter every pirate, the more safeguards you use, and the greater the variety you use, the more difficult the pirate’s task. Eventually it makes more sense for potential pirates to either purchase your product or pirate a different, less secure package.

Hiding Calls

A pirate may analyze your object code and examine addresses referenced by CALL instructions to find the calls to the SentinelSuperPro interface routines. The pirate could then analyze the code of the interface routine and the code following each call in order to defeat the lock.

One method to avoid detection of your queries is to call the SentinelSuperPro without using the assembler language CALL instruction. Instead, push the return address onto the stack followed by the procedure address, and then execute a RET (return) instruction.

Inserting Extra Data

Analysis of your code can also be made more difficult by inserting frequent “garbage” data bytes. This process is effective at throwing static disassemblers “out of sync.”

For example, after each unconditional jump and return, insert a garbage data byte or two whose value is equal to the first byte of a very long assembly language instruction.

This same technique can be used following conditional branches, as long as the preceding code always guarantees the branch is invoked. Such a jump or branch may also be used immediately prior to the call, with an intervening data byte.

Encryption Techniques

Another effective method for protecting your application is to use reversible encryption techniques to encrypt and decrypt data.

To do this, you can use the *sproQuery()* function to scramble a data string, and then use the scrambled response string to encrypt your application code. You would then ship your application to the field with encrypted code, which is decrypted only if the hardware key is attached to the parallel port or USB port.

Most encryption algorithms depend on a key value, sometimes called a password or seed, to transform the data. Some encryption algorithms are reversible: they can be used to decrypt what is encrypted.

Using a different seed produces different encrypted results, but reproduces the original data if that seed is also used for decryption.

Note that these techniques may be difficult or impossible to implement in some languages.

Using Returned Values as Encryption Seeds

You can use SentinelSuperPro returned values as encryption seeds to disguise critical portions of data or code as random data until decrypted for use. If the encryption seed is derived from values produced by the SentinelSuperPro, the correct hardware key must be present before the code can be decrypted and executed.

When decrypted data is "in the clear," use some other form of protection to block interrupts that are used by debuggers to gain control.

The most common reversible algorithms use the Boolean operator EXCLUSIVE OR (XOR). XOR works as follows:

- If a seed bit has a value of 1, XOR reverses the state of the corresponding bit in the original string and copies it to the result.
- If a seed bit has a value of 0, XOR copies the corresponding bit in the original string to the result.

Applying the same algorithm to the result reverses the encryption and restores the data to its original state.

Example

The following example uses the XOR operator to encrypt a 16-bit hex number (8FA3) using the seed 4B6A.

Hex	Binary	Description
8FA3	1000111110100011	data to encrypt
4B6A	0100101101101010	seed
	--------------------------	XOR algorithm
C4C9	1100010011001001	encrypted result

Note that everywhere a bit in the seed is 1, the result bit is the opposite state of the data bit. Where the seed contains a 0, the result bit is the same as the data bit. Without knowing the seed, the encrypted result is meaningless.

To reproduce the original data, apply the XOR algorithm to the encrypted result using the same seed, as shown below.

Hex	Binary	Description
C4C9	1100010011001001	encrypted result
4B6A	0100101101101010	seed
	---------------------------	XOR algorithm
8FA3	1000111110100011	original data

Using Longer Encryption Seeds

If the data to encrypt is longer, a longer seed can be constructed. The scheme for forming such a seed may be as complicated as you wish.

For example, the number 4B6A can be expanded to a 32-byte string by “rotating” it left 15 times and stringing the results of each rotation together. This yields the following hex string:

```
4B6A 96D4 2DA9 5B52 B6A4 6D49 DA92 B525
6A4B D496 A92D 525B A4B6 496D 92DA 25B5
```

You can use this string as a seed with the XOR algorithm to encrypt a 32-byte string.

For example, the ASCII string “This is the secret of my program” can be represented as the following hex string:

```
5468 6973 2069 7320 7468 6520 7365 6372
6574 206F 6620 4D59 2070 726F 6772 616D
```

Using the 32-byte seed with the XOR algorithm produces the following encrypted result:

```
1F02 FFA7 0DC0 2872 C2CC 0869 A9F7 D657
OF3F F4F9 CF0D 1F02 84C6 3B04 F5A8 44D8
```

The result looks nothing like the original character string, yet the original data can be recovered easily using the same algorithm and seed that changed it.

You can use this method with entire sections of code within your application, expanding the seed as needed.

Using Advanced Encryption Techniques

You can make encryption even more complex, depending on how sophisticated you want to make your application. For example:

- Use values returned by the SentinelSuperPro key as seeds for a pseudo-random number generator that generates seed encryption patterns.
- Use returned values to decrypt subroutines that decrypt code using an entirely different encryption method and seed.
- Instead of using the XOR operator, multiply each byte by a seed to encrypt it. Divide by the same seed to decrypt the data.

  Multiplying an 8-bit value by an 8-bit value yields a 16-bit result. The result is double the size of a data string produced with the XOR operator, but is also harder to crack. If you use this technique, make sure your multiplier/divisor seed does not equal 0.

Your local technical library should have several reference materials on encryption. Other topics you may wish to investigate include codes, cryptology, and the National Security Agency (NSA).

Additional Strategies Using Data Words

The following are some additional strategies you can use with cells programmed as read/write data words (access code 0) or read-only data words (access code 1).

- • Word 1 contains your unique, read-only developer ID. This number cannot be duplicated. Program your application to read this cell at least once.
- • Store machine code in data words. This code can be read, checksummed, and executed in a way that is verified by a different part of the application.
- • Program the application’s serial number into a data word. Read the cell and compare the value to the correct serial number.
- • If you have multiple application packages, store the serial number for each in a separate data word.
- • Store the user’s name in data words as ASCII bytes, then compare or display it.
- • Program data words with pseudo-random numbers that serve as part of a decryption key used by the program to access secure information.
- • Program data words with pseudo-random numbers based on the program’s serial number, then have the application verify them.
- • Use the 56 programmable cells as one large, 896-bit bitmap. Various combinations of bits can determine features or other responses, depending on your application.

Using Stepped Access

The SentinelSuperPro can be used to control access to features within a single program, based on criteria you specify. This is called “stepped access.”

Stepped access is useful if you market multiple versions of an application. For example, you may offer a basic package, an expanded package with some added features, and a deluxe package with all features.

Using stepped access, the program would contain an array of conditions instructing the system to activate different features based on the value returned by the SentinelSuperPro. By using a different algorithm descriptor for each package, you can control the features implemented by the software.

Three algorithm descriptors will produce three different values for the same string. For the string ABCD, for example, three algorithm descriptors might produce the values 2610, 1830, and 6287.

Your program would contain statements that produce different responses based on the returned value, as illustrated by the following pseudo-code:

```
IF result EQUALS
2610 THEN enable basic features
1830 THEN enable basic+expanded features
6287 THEN enable all features
```

If you have many steps or conditions, they can be stored in an array. The program would check the array for a match with the string and return the number of the element matched. This number then determines the features activated or the action taken by the program.

Querying Counter Words

You can use the *sproQuery()* function to query a counter word to verify that it has been counted down. This technique is useful in that it lets you see if a pirate is trying to side-step your counter, in an attempt to continue the demo condition indefinitely.

Counters used in this way must be two words long:

- The first word has an access code of 2 (counter). This word must be an even-numbered cell.

- The second word has an access code of 3 (algorithm/hidden). This word must be an odd-numbered cell.

You code your application to send a query to the counter/algorithm descriptor before doing each decrement. A given query string should return a *different* response value after each decrement, because the value in the first word (the counter) will have changed. If the query returns the *same* value, the counter was not decremented, which may mean that a pirate is attempting to circumvent your protection scheme.

If the Hardware Key Is Not Attached

If no key is connected to the computer, an error is returned by the *sproFindFirstUnit()* function. If connection is established but the key is later removed, subsequent API calls return errors.

If your program detects that the SentinelSuperPro key is not present, it is up to you to decide what action to take. Possible actions include the following:

- Display a message and wait for the user to press a key. This method does not prevent users from running the application, but makes doing so extremely annoying, especially if the application queries the hardware key frequently.
- Abort the application after a predetermined number of failed queries.
- Allow the application to appear as if it is functioning properly while in fact it is not. (Be very careful if you use this method. Less drastic actions should be considered first.)
- Display a critical error message and tell the user to contact your technical support department.

These are just a few suggestions. You can implement any combination of them to suit your needs and your application.

All attempts have been made to guarantee error-free transmission. However, a small possibility exists that an invalid response may be received even though the correct key is attached. If you receive an invalid response, always retry the query one or more times. If the response is consistently invalid, take the action you deem appropriate.

Appendix A - API Status Codes

Table A-1 describes the status codes the API library may return to your application.

For programming details specific to your language, refer to your language's API.TXT file and the include files.

Note: Status information can be obtained only from system drivers. If a linked-in or DLL driver is being used, these calls always return status 30 (API status unavailable).

Table A-1. API Status Codes

Status Code (Decimal)	Description
0	**success -** The call completed successfully.
1	**invalid function code -** You specified an invalid function code. See your language's include file (e.g., SUPERPRO.H) for valid API function codes. Generally, this error should not occur if you are using a Rainbow-provided interface to communicate with the driver.
2	**invalid packet -** A checksum error was detected in the command packet, indicating an internal inconsistency. The packet record or UNITINFO structure has not been initialized or may have been tampered with. Generally, this error should not occur if you are using a Rainbow-provided interface to communicate with the driver.
3	**unit not found -** Either *sproFindFirstUnit()* or *sproFindNextUnit()* could not find the specified SentinelSuperPro key. Make sure you are sending the correct developer ID. This error is returned by other functions if the key has disappeared (that is, has been unplugged).
4	**access denied -** You attempted to perform an illegal action on a word. For example, you may have tried to read an algorithm/hidden word, write to a locked word, or decrement a word that is not a data or counter word.

Table A-1. API Status Codes (continued)

Status Code (Decimal)	Description
5	**invalid memory address -** You specified an invalid SentinelSuperPro memory address. Valid addresses are 0-63 decimal (0-3F hex). Cells 0-7 are invalid for many operations. Algorithm descriptors must be referenced by the first (even) address.
6	**invalid access code -** You specified an invalid access code. The access code must be 0 (read/write data), 1 (read-only data), 2 (counter), or 3 (algorithm/hidden).
7	**port is busy -** The requested operation could not be completed because the port is busy. This can happen if there is considerable printer activity, or if a unit on the port is performing a write operation and is blocking the port. Try the function again.
8	**write not ready -** The write or decrement could not be performed due to a momentary lack of sufficient power. Try the operation again.
9	**no port installed -** No parallel ports could be found on the workstation.
10	**already zero -** You tried to decrement a counter or data word that already contains the value 0. If you are using the counter to control demo program executions, this condition may occur after the corresponding algorithm descriptor has been reactivated with its activation password.
12	**driver not installed -** The system device driver was not installed or detected. Communication to the unit was not possible. Verify that the device driver is properly loaded.
13	**communications error -** The system device driver is having problems communicating with the unit. Verify that the device driver is properly installed.
18	**version not supported -** The current system device driver is outdated. Update the driver.
19	**OS environment not supported -** The operating system or environment is not supported by the client library. Contact Technical Support.
20	**query too long -** You sent a query string longer than 56 characters. Send a shorter string.
30	**driver is busy -** The system driver is busy. Try the operation again.
31	**port allocation failure -** Failure to allocate a parallel port through the operating system’s parallel port contention handler.

Table A-1. API Status Codes (continued)

Status Code (Decimal)	Description
32	**port release failure -** Failure to release a previously allocated parallel port through the operating system’s parallel port contention handler.
39	**acquire port timeout -** Failure to acquire access to a parallel port within the defined time-out.
42	signal not supported **-** The particular machine does not support a signal line. For example, an attempt may have been made to use the ACK line on a NEC 9800 computer.
57	init not called **-** The key is not initialized. Call the *sproInitialize()* function before calling the function that generated this error.
58	driver type not supported **-** The type of driver access, either direct I/O or system driver, is not supported for the defined operating system and client library.
59	fail on driver comm **-** The client library failed on communicating with the Rainbow system driver.
60	API status unavailable **-** The extended API Status function is unavailable.
255	invalid status **-** An invalid status code was returned.

Appendix B - SPROX Error Messages

Table B-1 describes the error messages that may appear when you run the SPROX driver encryption utility described in Chapter 6.

Table B-1. SPROX Error Messages

Message	**Description**
B-00 : Not a SuperPro Object file or corrupt file => <filename>	The input file you specified is not an object file, uses an unknown format, or has been corrupted. Enter the name of the SentinelSuperPro driver object code file.
B-01 : Unable to locate file => <filename>	The input file you specified could not be found; check the pathname. This error also appears if the file you specified is marked as read-only. You may also have left the input or output file name blank; file names must be provided.
B-02 : Invalid encryption method => <entry>	You entered a letter other than X, A, R, or S for the encryption method. Type the first letter of the method you want to use.
B-03 : Invalid encryption seed => <value>	The encryption seed must be between 1 and 255. Enter a valid value.
B-04 : Range exceeds the end of the file => <calculated range>	The starting byte plus the byte count exceeds the end of the input file. If you want to encrypt to the end of the file, specify 0 for the byte count (/N0 on the command line).
B-05 : Invalid numeric value => <entry>	At least one of the numbers you entered is not a number or exceeds the maximum size for integers. Enter valid numbers within the displayed ranges.
B-06 : Invalid encryption seed modifier => <value>	The encryption seed modifier must be between 0 and 255. Enter a valid value. If you do not want the seed value changed during encryption, enter 0.
B- 07 : Swap encryption requires an even number of bytes => <value>	You selected SWAP encryption, but the byte count you entered is odd. The second number in the encryption range must be even. Change the byte count or select a different encryption method.

Table B-1. SPROX Error Messages (continued)

Message	Description
B-08 : File too big => <filename>	The object code file you specified is too large to encrypt with SPROX. This tool is designed to encrypt the SentinelSuperPro driver only — it is not a general-use encryption tool. Enter the correct file name.
B-09 : Unable to create file => <filename>	SPROX could not create the specified output file. Make sure the pathname is valid and any directories specified in the pathname already exist. Also, make sure you have adequate disk space available.
B-10 : Options {/O,/M,/S, /A,/B,/N} also require option /I (Input File)	If you specify any of these parameters on the SPROX command line, you must also specify the input file name. Reenter the command line and include /I followed by the name of the driver file to be encrypted. Alternatively, omit all of the parameters and run SPROX interactively.
B-11 : Output file already exists => <filename>	The file name specified (or defaulted to) by /O already exists, but you did not enter /V (overwrite) on the command line. To replace the existing file, reenter the command line with /V. Alternatively, select a new output file name.
B-12 : Process complete, but unable to create Log File => <filename>	SPROX could not create the log file. Usually, this indicates that the log file name is not a valid DOS pathname. Make sure you specify a valid pathname using a directory that already exists. Also, make sure you enter the file name directly after /L, with no intervening punctuation.
B-13 : Process complete, but error writing to Log File => <filename>	SPROX could not write to the log file. This may mean that the file is marked as read-only, or that you do are out of disk space.
B-14 : Not a SuperPro Object file or corrupt file => <filename>	The input file you specified is not an object file, uses an unknown format, or has been corrupted. Enter the name of the SentinelSuperPro driver object code file.
B-15 : Range requires 2 numbers separated by a space, Ex: 42 1001	You did not enter two numbers separated by a space. Enter the first byte to encrypt, a space, then the byte count. Do not enter letters or punctuation. You can omit the second number if you want to encrypt to the end of the file.
B-16 : Invalid starting offset for encryption range => <entry>	The first number you entered for the encryption range is not within the valid range displayed. Enter a number between 0 (the first byte in the file) and the second number shown in the range (the last byte in the file).

Table B-1. SPROX Error Messages (continued)

Message	Description
B-17 : Invalid number of bytes for encryption range => <entry>	The second number you entered for the encryption range is not within the valid range displayed. Enter a number between 0 and the second number shown in the range (the last byte in the file). Note that using 0 to represent the end of the file is recommended, as you will not have to modify your code if the driver size changes.
B-18 : Swap encryption must start before last byte of file	You specified SWAP encryption and a starting byte that is the last byte of the file. SWAP encryption requires a minimum of two bytes. Select a different encryption method, or specify an earlier start byte.
B-19 : Not a SuperPro Object file or corrupt file => <filename>	The input file you specified is not an object file, uses an unknown format, or has been corrupted. Enter the name of the SentinelSuperPro driver object code file.

Appendix C. Programming Worksheet

Below is a worksheet you can photocopy and use as an aid while designing your protection scheme and programming your keys.

Rainbow Technologies, Inc.
SentinelSuperPro Programming Worksheet

	0/8	1/9	2/A	3/B	4/C	5/D	6/E	7/F
00	**** SN	**** DI	**** OP	**** OP	**** WP	**** RW	**** RW	**** RW
08								
10								
18								
20								
28								
30								

38								

Appendix D - Quick Reference Tables

Table D-1. Restricted Cells

Cell(s)	Contents	Readable?
0	Randomly assigned serial number	Yes
1	Developer ID; unique to your company/product	Yes
2	Overwrite password word 1	No
3	Overwrite password word 2	No
4	Write password	No
5-7	Reserved for future use by Rainbow	No

Table D-2. Access Codes

Code	Description
0	**Read/write data word.** Your application can read the cell and, if the write password is supplied, modify its contents.
1	**Read-only (locked) data word**. Your application can read the cell but cannot change it without the overwrite password.
2	**Counter word.** The cell contains a value that your application can decrement using the write password. The value cannot be reset without the overwrite password.
3	**Locked and hidden/algorithm word.** Your application cannot read the cell’s value, and cannot change it without the overwrite password.

Table D-3. Algorithm Word Values

	Word 1	Word 2	
		Enhanced Engine Disabled[1]	Enhanced Engine Enabled[1]
Algo Inactive	0000-FFFFh	0000 to 3FFFh	4000 to 7FFFh
Algo Active	0000-FFFFh	8000 to BFFFh	C000 to FFFFh

[1] *Enhanced algorithm engine (64-bit) for maximum security.*

Table D-4. Cell Types

Cell Type	Access Code	Description
**	0	Undefined
AA	3	Active Algorithm
AH	3	Algorithm Half
AP	3	Algorithm Password
CA	2	Algorithm Counter Word
CW	2	Counter Word
DI	1	Developer ID
DL	1	Locked Data Word
DW	0	Data Word
IA	3	Inactive Algorithm
OP	3	Overwrite Password
RW	3	Reserved Word
SN	1	Serial Number
WP	3	Write Password

Table D-5. Valid Addresses For Algorithm Words

Algorithm Method	Valid Starting Cell Addresses
Algorithm (2 cells: algo, algo)	`08, 0A, 0C, 0E, 10, 12, 14, 16, 18, 1A, 1C, 1E, 20, 22, 24, 26, 28, 2A, 2C, 2E, 30, 32, 34, 36, 38, 3A, 3C, 3E`
Algorithm with password (4 cells: algo, algo, AP, AP)	`08, 0C, 10, 14, 18, 1C, 20, 24, 28, 2C, 30, 34, 38, 3C`
Algorithm with counter (3 cells: CA, algo, algo)	`0B, 13, 1B, 23, 2B, 33, 3B`
Algorithm with two counters (4 cells: CA, CA, algo, algo)	`0A, 12, 1A, 22, 2A, 32, 3A`
Algorithm with password and counter (5 cells: CA, algo, algo, AP, AP)	`0B, 13, 1B, 23, 2B, 33, 3B`
Algorithm with password and two counters (6 cells: CA, CA, algo, algo, AP, AP)	`0A, 12, 1A, 22, 2A, 32, 3A`

Table D-6. Advanced Editor Item Types

Item Type	Description
Algorithm (2 cells)	A simple algorithm descriptor. Use this type if you want to scramble an input string but do not want an associated counter or activation password.
Algorithm with Counter (3 cells)	An algorithm descriptor that has a counter associated with it. Usually, this item type is used to limit the number of times a demo program can be executed.
Algorithm with Counter and Password (5 cells)	An algorithm descriptor that has both a counter and a password associated with it. Use this type if (1) the algorithm is to be deactivated when the counter reaches 0, and (2) the user must input a password to get the application to run initially, or to reactivate it after the counter reaches 0.

Table D-6. Advanced Editor Item Types

Item Type	**Description**
Algorithm with Password (4 cells)	An algorithm descriptor that has a password associated with it. Use this type if you want the user to enter a password to make the application run initially.
Algorithm with Two Counters (4 cells)	An algorithm descriptor that has two counters associated with it. The first counter that reaches 0 deactivates the algorithm, which usually stops your application from executing properly.
Algorithm with Two Counters and Password (6 cells)	An algorithm descriptor that has two counters and a password. Use this type if (1) the algorithm is to be deactivated when either counter reaches 0, and (2) the user must input a password to make the application run initially, or to reactivate it after it is deactivated.
Algorithm Word (1 cell)	A cell that will be used as half of an algorithm descriptor. Use this type if you want the user or installer to supply the second half of the algorithm descriptor in the field. Make sure you are aware of all rules governing algorithm descriptor values and location.
Counter (1 cell)	A cell that contains a value you can decrement. Use this type to limit the number of times a demo program can be run, or to limit the number of times any particular operation is performed.
Data Word (1 cell)	A cell that contains a data value your application can test (and change) during execution. For example, you might store a serial number or feature control code.
Locked Data Word (1 cell)	A cell that contains a read-only data value your application can test during execution. Your application can read the stored data but cannot change it without the overwrite password.

Table D-7. API Functions

Function	Description
sproActivateAlgorithm()	Activates an inactive algorithm descriptor so that it can be used by the *sproQuery()* function.
sproCfgLibParams()	Lets you configure various aspects of the SentinelSuperPro interface.
sproDecrement()	Decrements a counter word by 1. If the counter is associated with an algorithm descriptor, decrementing the counter to 0 deactivates the algorithm.
sproExtendedRead()	Reads the value and access code of any unhidden word in the key.
sproFindFirstUnit()	Searches all attached keys for a specified developer ID.
sproFindNextUnit()	Searches for the next key with the same developer ID.
sproGetVersion()	Returns the SentinelSuperPro driver's version number.
sproInitialize()	Performs any required initialization. This function must be called once before any other API function is called.
sproOverwrite()	Changes the value and/or access code of any word except the reserved words in cells 0 through 7.
sproQuery()	Sends a data string to the key, scrambles it using a specified algorithm descriptor, and returns the scrambled string to the application.
sproRead()	Reads the value of any unhidden word in the key.
sproWrite()	Changes the value and/or access code of any word with an access code of 0 (read/write data).

Note: For OS/2, Windows NT, and Windows 95/98, each function name is preceded by RNBO.

Index