2 Introduction

2.1 Hardlock Documentation and Manuals

The Hardlock documentation is broken down as follows:

- **Hardlock Technical Manual**
  Introduces the entire software protection palette and defines its areas of applications. Describes CP.EXE programming software and provides general information on encryption and Hardlock hardware.

- **HL-Crypt, Automatic Implementation for Hardlock**
  Identifies and describes the various ways of automatically integrating software protection.

- **Hardlock Bistro, the visual Hardlock program development environment**
  Describes the few steps to your Hardlock-protected application.

- **HL-Server, Licensing Network-Wide**
  Describes how Hardlock can be used in networks.

- **Hardlock API, Manual Implementation**
  Identifies and describes the various ways of manually implementing software protection.

- **Hardlock Terms**
  Explains key Hardlock terms and concepts. Cross-references are denoted with an arrow. For example:

  ➔Sub-code

Manuals can also be purchased individually.

2.2 A Few Opening Remarks...

To achieve effective software protection, you must always keep in mind the principle of the weakest link in a chain. In sophisticated software protection systems such as Hardlock, the "links" of the hardware "chain" (copy protection, protected coding, algorithmic protection, etc.) are extremely strong and reliable.

With the Hardlock software protection system, the weakest link in the protection chain is the actual implementation of the module in the program to be protected.

When implementing Hardlock in your software, the objective is to couple the program and protection hardware (Hardlock) to create a single, indissoluble unit. Generally speaking, the application must query the Hardlock and use the module functions provided.

This may sound simple, but in reality creating effective software protection takes time. In fact, the degree of protection afforded by a manual implementation is directly proportional to the amount of time invested. That you must have sufficiently complex algorithms to begin with goes without saying.

Unfortunately, there are (still) many software manufacturers, particularly in the area of data protection, who offer utterly primitive protection methods (for example, constant XOR ciphers). Although two sets of encrypted text may at first glance appear equally garbled, there may be great differences in the complexity of their encryption. Data encrypted by inverting a few bits and data encrypted using a sophisticated encryption method may look the same, but they have completely different cryptographic behavior.

Faced with deadlines and other constraints, experienced programmers often do not have the time to develop an effective strategy for implementing software protection. More often than not, this task is ignored and neglected. However, the earlier the implementation of software protection is considered in a project, the easier and more effective implementation is in the end.

To achieve optimal protection, you must be very familiar with your compiler, program, operating system and, of course, the Hardlock itself. Several days are required to achieve truly effective manual protection.

For all DOS, Windows and Win32 programs, we recommend HL-Crypt. This system lets you implement effective software protection within minutes. Thanks to years of experience, we know how a potential hacker goes about gaining access to your program. Using this knowledge, we developed HL-Crypt, a truly effective system for automatically implementing software copy protection. For more information, please see the documentation for automatic implementation with HL-Crypt.

☞ Note:

Use the Latteccino program in the new Hardlock Bistro software to test the API functions. The software runs under Windows 95 or Windows NT only. For more information, please refer to the Hardlock Bistro manual.

The Latteccino program is an intuitive interface to the API. You only need to refer to this manual if you want to find out more about the individual API function.

3 API Operation

3.1 Application Programming Interface (API)

3.1.1 General Definition

The ➔API (**A**pplication **P**rogramming **I**nterface) for Hardlock is the interface between the application you want to protect and the Hardlock. It contains all functions required for controlling and querying a Hardlock (e.g. as a Hardlock library). The API is divided into two levels:

The ***low-level API*** contains all hardware and operating system-dependent routines for controlling the Hardlock. The low-level routines of the API are always available as libraries or object files (also see "Naming Conventions for API Objects").

The ***high-level API*** was designed to make working with the API easier. It is available for compilers in the most popular high-level languages. The high-level API functions described in the manual also support the programming interface of the low-level API. Implementation of the high-level API is provided by the source code.

3.1.2 Programming with the Hardlock API

The main advantage of the high-level functions is that you only have to define the parameters required for the application when calling them. All other internal parameters required for addressing the Hardlock are automatically managed by the high-level API and passed on as a complete "structure" to the low-level API together with the parameters relevant to the application.

For most applications you will only need the high-level functions. However, we would be happy to send you a complete description of the low-level routines should you wish to use the programming interface of the low-level API. Excerpts from this documentation are also available as online text in the API debugger TESTAPI.EXE.

☞ Note:

> Hardlock API is used in all protection products (manual implementations, HL-Server, HL-Crypt). Be sure to only use programs with the same API main version number to ensure that the various products work together properly. For example, an HL-Server with API version 2.xx is not compatible with an application which was manually implemented using API version 3.xx. Products with the same main version number (e.g. 3.xx) can be used together without any problem.

3.1.3 Using High-level Functions

For information on how to use high-level language calls, please read the detailed descriptions of the individual functions provided in Chapter 6 and refer to the example programs for the various compilers provided on the Hardlock CD.

Compiler settings, objects and libraries required for your particular program are contained in batch data and make files included. These are generally for a minimum configuration. Before integrating Hardlock in your software, first try to compile the example program.

We emphasize that the example programs are only meant to clarify how the various function calls are used and how they operate. We have kept them as simple and concise as possible. They are by ***no means exemplary of an effective manual implementation***. Do not simply transfer the examples to your program! Remember, a potential hacker also has access to our sample routines.

3.2 What Can Hardlock Do For You?

3.2.1 Algorithmic Protection

Hardlock operates based on the principle of algorithmic protection. This means a mathematical function is incorporated in the hardware of the Hardlock. The behavior of this function depends on which coding was programmed into the Hardlock. Hardlocks with the same parameters but different coding deliver different results.

3.2.2 Encrypting with Hardlock

Used together with the implemented function, Hardlock serves as a key generator. The key is hidden in the hardware. This eliminates the need to keep the key confidential or store it in a safe place. With the Hardlock, constants, data and code segments of an application can be decrypted and encrypted while the program is running.

3.3 HL_CODE(...) Developing Tool

The HL_CODE(...) function contains the cryptographic algorithm K-EYE. Used together with Hardlock, this algorithm encrypts data into 64-bit (8-byte) blocks (block ciphers). Ciphers operate in both directions, i.e. by using the same Hardlock, the cipher text can be transformed back into plain text. The HL_CODE(...) must be supplied with a pointer for a certain data area and the number of encrypted 8-byte blocks. Up to 64 KB = 8192 blocks can be encrypted with a single call.

Encrypting several data blocks (of 8 bytes each) with a single call of HL_CODE(...) leads to different encryption results than encrypting these data blocks by calling the function several times (for example, for each data block or several groups of a few data blocks).The example below should help clarify this procedure:

Example:

```
Program 1:                 Program 2:

Byte Block1 [8];           Byte Block1 [8];
Byte Block2 [8];           Byte Block2 [8];

HL_CODE(Block1,1);         HL_CODE(Block1,2);
HL_CODE(Block2,1);
```

In both cases, two 8-byte blocks are encrypted, but the final results of encryption are different.

3.3.1 Speed of HL_CODE(...)

HL_CODE(...) with a block length of 1 encrypts at a rate of about 8 KB per second (on a parallel port) on a 20 MHz 486 SX computer. This is the physical speed with which the bits are encrypted by Hardlock. However, this speed is not actually achieved since API performs additional testing and protection processes to protect the application from losing data during encryption and if a Hardlock is not available. If HL_CODE(...) is called with a greater number of blocks, not all bits are sent to Hardlock. This ensures faster encryption of large volumes of data. For example, 64 blocks (=512 bytes) are encrypted about 10 times faster.

Increasing the number of blocks with HL_CODE(...) reduces the ratio of key information generated by Hardlock to the volume of encrypted data. However, this reduction in cryptographic protection is secondary compared to the increase in speed. Changing keys every 512 bytes, for example, does not significantly jeopardize the protection of data encryption.

4 Manual Implementation

At this point, we assume you've decided to go ahead with manual implementation and are prepared to immerse yourself more deeply in the subject of cryptographic software protection. Please keep in mind that it is basically up to you how effectively your "investment" is protected. It is your choice: you can either quickly (and not very safely) implement software protection or you can take your time and develop a sound, well thought out implementation. We also recommend using HL-Crypt as an additional tool when manually protecting an application.

4.1 Strategic Considerations

To effectively protect your application through manual implementation, you should incorporate appropriate protection mechanisms in your application right from the start. When developing your protection concept, keep the following factors in mind.

4.1.1 Tools of the Hacker

Let's first look at how hackers operate. First off, they generally use a debug program. At the least, they have the DEBUG.EXE that comes with MS-DOS. However, to be on the safe side, let's assume that a hacker has the best "tools" available. This includes debuggers that use a 386/486 processor in protected mode or special debug hardware. Such debuggers permit a program to be taken apart and analyzed step by step. A hacker has to go through the assembler code generated by your compiler and find all inquiries made to the Hardlock. After he has recognized the critical protection measures, he has to modify the program so that it operates normally without any protection.

Ultimately, it is not a single, particularly sophisticated measure that prevents a hacker from gaining access to your program, but rather a multitude of subtle, simple measures which eventually exhaust his efforts.

4.1.2 Can a Program be 100 % Safe?

Unfortunately, the answer to this question is "No.” Any supplier of software protection systems who claims that his products offer absolute protection cannot be taken seriously. Hardlock protection measures let you increase protection - and thus the time and effort required for analyzing the program - to any degree desired. At some point, time is money for even the best and most determined of hackers. But the bottom line is: Any program protection system can be recognized, analyzed and circumvented with enough time, money and determination.

4.1.3 Generating Code, Data and Parameters with HL_CODE(...)

With Hardlock you can easily protect programs so that it is virtually impossible to circumvent the protection system ***without the right Hardlock.*** The parameters, data and code segments required for running the program are useless until they are decrypted by Hardlock. To run the program without the Hardlock, a hacker has to replace all encrypted data with decrypted data. Since the hacker cannot simply guess what the missing information is, he has to have access to the Hardlock.

If data and parameter encryption was frequently repeated and distributed throughout the entire program, a hacker would require an enormous amount of time and effort to record all "correct” results and incorporate them in the program ("➔Patching"), even with the appropriate Hardlock (which we assume he has). This process can be prolonged indefinitely by incorporating "fake assumptions” and consistency checks.

4.1.4 Time Window Method

Data required can be decrypted as needed while the program is running. This way certain information remains encrypted in the program except for when it is actually being used. This information is never completely decrypted and available as plain text all at once.

4.2 Implementation Tips

Depending on the type of compiler being used, some of methods described here cannot be used directly with your programming language. However, many compilers permit direct access to your program code by using, for example, online assembler instructions or linking modules of machine-oriented program sections.

Please remember that there is no patent solution for safe manual implementation. We can only provide you with suggestions and tips based on our experience, and the necessary tools and assistance in answering your questions. However, the ultimate degree of manual protection is up to your imagination and willingness to invest time in the quality of your protection.

4.2.1 Hardlock Initialization at the Program Start

You should integrate a simple Hardlock query during Hardlock initialization in the beginning of your program. This not only ensures that the Hardlock with the correct ➔module address is available, it also checks whether the Hardlock found with this module address has the right coding (➔base code ***and*** ➔sub-code). Hardlocks with the same module address but different encryption behavior can be used.

Both tasks, that is, checking whether the Hardlock is available and identifying the encryption behavior, are performed by the HL_LOGIN(...) function with the appropriate parameters.

These preliminary tasks merely serve to initialize Hardlock access and correctly recognize the Hardlock. They do not actually protect the software. However, once they have been performed, the system can be sure that the Hardlock with the expected coding is available.

Please keep in mind that it is very easy for a hacker to get around a software protection system that only queries at the program start. Moreover, such a system also offers no protection against the application being used at different work stations.

4.2.2 Protection While Program is Running

In developing a protection strategy, you should basically aim to create a complex network of protective measures (most of which are just fakes to lead the hacker astray) which branch out throughout the program, rather than incorporate a few, particularly sophisticated individual measures. Hardlock queries should be scattered throughout all levels of the menu structure.

4.2.3 Physical Separation

Program sequences designed to query the Hardlock should not always be programmed consecutively since this (almost) always results in (nearly) identical code. This makes it easier for a hacker to locate Hardlock queries. Physically separate these sequences in your source code (as much as possible).

4.2.4 Force the Hacker to Get to Know the Program

In most cases, a hacker does not work with the program himself and is not familiar with how it is used. Imagine how long it would take a young hacker to become familiar with, for example, a complex accounting program down to the last dialog. It will always be difficult for him to make a correct entry and make sure that the program is running correctly. Protected programs also pose a psychological barrier to hackers since they never can be sure if they have really eliminated all traps.

4.2.5 Generate Noise

Generate noise in encrypted data by incorporating many Hardlock queries or decryptions and encryptions with HL_CODE(...). Design the system so that irrelevant values are given as arguments that are produced by random number generators, time values, intermediate results of calculations, etc. Of course, these queries should not lead to any meaningful results. This makes it practically impossible to trace calls. The idea is to distract the hacker from the truly necessary data.

4.2.6 Encrypt Data and Parameters

When the program is running, do not decrypt parameters, data and code segments (caution when working in protected mode) until they are required. By being able to encrypt information while a program is running, you can keep information encrypted and only convert it into plain text when it is actually required (time window method). For example, use encrypted data to initialize loops or transfer parameters to a function. Do not save a decrypted copy of the data in your program for comparing decrypted values to original values. Storing encrypted values as local variables on the stack also makes debugging more difficult.

4.2.7 Encrypt External Data

Also encrypt external data required for your program, such as configuration files, serialization information, which is decrypted while the program is running.

4.2.8 Confirm Hardlock's Existence

Before large volumes of data are decrypted, have the system check whether the Hardlock with your coding is present to prevent any important data from being destroyed.

☞ Note:

> Specified data may be modified, i.e. incorrectly encrypted or decrypted, if the Hardlock is not present when the HL_CODE(...) function is activated. Your system may crash if it continues to process this incorrect data.

Use the HL_AVAIL() function to check if the Hardlock is available.

4.2.9 Incorporate Hardlock Memory Option

If your Hardlock is equipped with the Memory Option, you can also incorporate the individual memory registers in the protection queries. For example, you can file customer name and serial numbers in ➔ROM and display them while the program is running. Memory registers not being used can be assigned random numbers that your application queries and then ignores. Keep in mind that the ➔RAM should not be used for protecting your program since it can be written to by practically any application.

4.2.10 Terminating Hardlock

Make sure that when the program is terminated, the Hardlock is properly deinitialized. This should be done with the HL_LOGOUT() function which releases the memory being used and frees the login entry for access through the HL-Server.

4.2.11 Consistency Checks

A hacker must modify your program for it to be used without the Hardlock. This means that he has to replace certain byte sequences in your program. Consistency checks make a program much more difficult to crack. Consistency checks cannot be used often enough and should always be performed in different ways. For example:

- Calculate and check the various test sums of critical sections of the program. The test sums themselves should also be safeguarded against modification, for example, by distributing them among several variables.

- Calculate and check certain sections of the EXE file, overlays, etc.

- Check whether individual critical commands have been modified.

Use different algorithms for calculating the various test sums. For example, you can file constants in the Hardlock's memory (if available) that are also used for calculating test sums.

Checking Control Functions

Implementation of queries should be complex and occur at many different levels. Some routines should inquire whether the Hardlock is still available. Others should determine whether the query routines are still available and operational. The causal relationship of these consistency checks can be extended to any degree. One routine which was just decrypted may check whether another routine which should be encrypted is, in fact, encrypted. Making one routine dependent on the other significantly increases the degree of software protection.

Checking Execution Times

Compare the actual execution times of routines with the amount of time normally required. If a routine is being run in the single-step or trace mode of a debugger, it runs much slower. Thus, if a routine is running considerably slower than it should, this may be an indication that a debug program is being used. If this is the case, respond subtly so that the hacker is not aware that he has been "found out.”

4.2.12 Measures After Recognizing an Attack

If, through one of the methods described above or through consistency checks, you determine that your program is being attacked or has been modified, there are a number of ways you can make your program ineffective. Be sure to separate the cause and effect of a hacker’s action so that he does not exactly know what he did to be discovered. Below are a number of ways this can be done.

Delay

Delay of the causal reaction. The response to the hacker’s being "found out” should not occur until a certain amount of time has elapsed (from a few seconds to a few days by using the system date).

Concealment

Concealing the causal relationship. One small change can lead to changes in many other values, of which only one really initiates the actual measure.

Distortions

Let your program continue running, but incorrectly. Calculations lead to incorrect results that are not immediately apparent.

Restrictions

Concealment through restrictions that are difficult to recognize. The hacker does not realize until a considerable amount of time has passed that the program is only running in demo mode (for example, only a limited number of data sets, or no print-outs).

4.2.13 Other Things to Keep in Mind

Do not waste your time concealing the queries in your high-level language through "sloppy” programming. Many compilers optimize code so that even in such cases "clean” assembler code is generated.

Do not incorporate any switches in your program to deactivate querying for internal test purposes - no matter how secret they may be. With such a switch, the complete protection system can be deactivated with a single modification of the program.

Make sure the release version of your software is created without any debug information.

4.3 Working with HL-Server

When implementing protection manually with the API, it is easy to differentiate between the querying of a local (connected to the computer) and remote (available through the HL-Server) Hardlock. Depending on what mode you selected (through HL_LOGIN(...)), the system will try to access a local Hardlock and/or one in the network. HL-Server must be installed in the network to address a Hardlock in the network.

By appropriately programming the system, with one program version you can use a local and network Hardlock automatically.

There is basically no difference between accessing a local Hardlock or a network Hardlock made available through the HL-Server (remote Hardlock). It is important, however, that you are aware of the different possibilities for use of local and remote Hardlocks. For more detailed information on using Hardlock in networked systems, refer to your HL-Server manual.

4.3.1 Modifying Access Type

Once a connection has been established to a Hardlock through HL_LOGIN(...), this connection remains intact until the HL_LOGOUT() function is activated. Use the following sequence to reinitialize from a local to a remote Hardlock:

```
...
HL_LOGOUT()
HL_LOGIN(..., REMOTE_DEVICE, .., ..)
...
```

For example, if a local Hardlock is removed you can automatically switch to a Hardlock provided by the HL-Server.

You should design your program so that it automatically switches initialization to a local Hardlock. This ensures that your system continues to operate correctly if, for some reason, the HL-Server (or remote Hardlock) is no longer available.

4.3.2 Timeout Behavior of Application

When working with HL-Server, you can also use a timeout value (see corresponding sections in the HL-Server manual). If no Hardlock query is sent to the HL-Server within the specified amount of time (for example, because your program is stuck in a loop without a Hardlock query and is waiting for a keyboard entry), the login entry of the station may be released again. This terminates the connection to the Hardlock for the protected program. A connection can easily be made again using the sequence above. However, in the meantime another station may have logged into the HL-Server and the license limit reached.

We recommend using a relatively high timeout value for installing the HL-Server or deactivating the timeout when working with manually protected programs. When protecting a program with the automatic implementation HL-Crypt, periodic Hardlock querying of the HL-Server can be ensured through background querying (see the HL-Crypt manual).

4.3.3 Sublicensing with Hardlock

Sublicensing is one way of managing and keeping track of individual program modules and functions. The difference between modules is defined by a "slot ID". In this case, a certain slot ID is allocated a number of available licenses.

Sublicensing with Hardlock is possible both on a local as well as a remote system (in conjunction with the HL-LiMa Server). Both systems are described in the following (the implementation of both options in your software is identical).

Local Sublicensing:

With a local Hardlock, only one license can be assigned per slot, meaning that the license is either released or not released for a particular slot (i.e., the program option is valid or invalid).

The relevant information is stored in the ROM area of the Hardlock Memory. This means that the Hardlock has to be coded, or "programmed", with the corresponding licensing information. The Cappuccino program is used to generate the memory content. On a local installation, the number of licenses is limited to 728 slots. Please note that licensing is only possible when the Hardlock used has a memory option, and, depending on the number of slots, memory area will be used for the sublicensing process.

Sublicensing with HL-LiMa Server:

In contrast to the procedure that applies to local Hardlocks, licensing by means of the HL-LiMa Server may configure a certain number of available licenses per slot.

The relevant information is communicated to the server in the form of an encrypted license file. This license file is also generated by Hardlock Bistro. The HL-LiMa Server uses this file to determine whether a login is permitted or rejected. For updating licensing, this means you can send users a file (e.g. via e-mail) or direct remote updates via the Internet.

Implementation:

For Hardlock implementation, the slot ID to be followed by the login must be known. It is passed to the server in the manual implementation with the function HL_LMLOGIN(…). The automatic implementation with HL-Crypt uses the option **sle:SLOTID**. The implementation can also be made from the Project Manager Espresso in Hardlock Bistro.

Example:

Module A uses Slot ID 1	No. of licenses:	3
Module B uses Slot ID 2	No. of licenses:	5
Module C uses Slot ID 3	No. of licenses:	2

Login for Program Module B:

```
HL_LMLOGIN(29809, "hardlock", ":?*@/f#y", 2, NULL)
```

5 Operating Systems

5.1 DOS

Under DOS the API is linked to the application through object files (.OBJ) or libraries (.LIB). The API becomes part of the actual application which can directly address the PC's hardware without any additional drivers.

5.2 Windows 3.x

Under Windows 3.x the API is also linked to the application through object files or libraries. However there are two different types of program links:

Static Linking

The required object files and libraries are directly "linked" to the program. This increases the size of the EXE file.

Dynamic Linking

The libraries are made available as DLLs (Dynamic Link Library). The required functions are not dynamically "linked" to the application until the program is running. Once the function is called the memory space is re-released.

Both the low-level core of the API and high-level routines can be linked through a common DLL. The DLL is capable of multitasking. You can start several Windows printer tasks and Hardlock tasks simultaneously, without them clashing

The DLL requires 4-8 KB for internal API data. In contrast to the "static linking" method, this is not taken from the data segment of the application since the DLL has its own data segment. An additional 300 bytes of the global heap is used for every active task.

Drivers: A driver must be installed to support Hardlock access under Windows 3.x. The installation of the HARDLOCK.VxD driver is easy with the INSTVXD.EXE installation routine. The driver can also be installed manually. Proceed as follows:

Copy the driver into the Windows 3.x system directory and enter the following line in the file SYSTEM.INI in the section [386enh]

```
device=c:\windows\system\hardlock.vxd
```

The driver is activated the next time Windows 3.x is started.

Do not forget to include the files INSTVXD.EXE and HARDLOCK.VxD in your software package.

5.3 OS/2

The information on the Windows DLL also applies to OS/2. In addition, the following points also apply:

DLL code has ***IOPL status***. Therefore, you must enter **IOPL=YES** in the CONFIG.SYS file. This is the default setting for OS/2 > 2.x. The system must also be able to access the OS/2 Hardlock DLL through the environment variable LIBPATH. This can be done, for example, by making sure LIBPATH contains the "." directory (stands for the current directory) and the DLL is in the same directory as the OS/2 program.

Three DLLs are available for OS/2 (version 1.3 and up):

1. HLOS2LOW.DLL: required for accessing the hardware (Hardlock) and must always be used. It contains the low-level part of the API. The DLL is written for OS/2 in 16-bit mode (IOPL). OBJ and

LIB libraries cannot be used with OS/2 since direct (i.e. without DLL), concurrent access of an OS/2 program to the printer port is not always possible.

2. HLOS2_32.DLL: provides a 32-bit program with the high-level interface in the form of a DLL and uses the low-level part in HLOS2LOW.DLL
3. HLOS2_16.DLL: provides a 16-bit program with the high-level interface in the form of a DLL and uses the low-level part in HLOS2LOW.DLL

As when working with Windows, the high-level API can be statically linked to the application.

The DLLs provided are capable of multitasking. You can start several OS/2 printer tasks and Hardlock tasks simultaneously without them clashing. However, since Hardlock accesses have priority, when Hardlock is intensively being accessed, other computer tasks (normal priority) are somewhat slower.

If several (DOS/Win) applications want to use the Hardlock and/or a local printer is being used, port status must be switched to "Shared Access" under OS/2. This is done as follows:

- Open the context menu of the printer icon (with the right-hand mouse button)
- Choose "Open" - "Settings"
- Choose "Output"
- Double-click the port being used (for example, "LPT1")
- Choose "Shared Access"

These settings are only required for protected DOS and Windows programs. They are not required for OS/2 programs which access Hardlock with the OS/2 DLL.

5.4 Windows NT

The driver (HARDLOCK.SYS) must be installed for accessing Hardlock under Windows NT.

The Hardlock API has offered full support of DOS and WIN16 programs under Windows NT since version 3.20. Programs linked with API versions < 3.20 can also be used, but are slower. The "Virtual Device Driver" HLVDD.DLL makes this possible. This file can also be used by your application as a 32-bit DLL.
Implementation ensures full Win32 (with s/c) support.

Use the HLINST.EXE program to install the drivers. This program is also included in the source code (all programs come with the Hardlock software).

Do not forget to include the files HARDLOCK.SYS, HLVDD.DLL and HLINST.EXE with your software (if applicable).

For more information on Hardlock and Windows NT (DLLs, driver installation, etc.) see the README files on your Hardlock CD.

5.5 Windows 95

Windows 95 monitors all direct access to the ports. This slows down Hardlock accesses. Hardlock API supplies the Virtual Device Driver HARDLOCK.VxD to get around this problem. This driver permits the system to address Hardlock directly without being interrupted by Windows 95. The driver also takes care of simultaneous print commands.

Installing the driver is easy. Simply copy the HARDLOCK.VxD file to the WINDOWS or WINDOWS/SYSTEM directory. You do not have to make any changes to the SYSTEM.INI file since the Hardlock API automatically loads the driver on request.

Do not forget to include the HARDLOCK.VxD file with your software.

5.6 DOS Extenders

Hardlock API supports a number of DOS extenders (both 16-bit and 32-bit).

When configuring your application or extender, make sure that 9 K of real mode memory is available to the API for using network routines. Data buffers for accessing the network are created here. Most DOS extenders are set as standard to 64 K of real mode memory. (For more details see the documentation of your DOS extender)

16-bit DOS Extenders

The current API version supports all 16-bit DOS extenders compatible with DPMI 0.9 or higher (for example, Windows 3.x DOS Box, Borland languages).

32-bit DOS Extenders

The current API version supports all DPMI-compatible 32-bit DOS extenders (for example, Rational DOS/4GW, PharLap 386|DOS).

AutoCAD12 uses the PharLap 386|DOS extender and is also compatible with Hardlock API. For more information on implementation please see the appropriate example program. The default setting for AutoCAD does not reserve any real mode memory. You must adjust the setting to use API network routines.

Example:

```
cfig386 acad.exe -minr 8192 -maxr 8192
```

5.7 UNIX

In order to use the API under UNIX, you have to regenerate the UNIX kernel (implementation of the API driver) since only it is authorized to access I/O addresses.

An API implementation (including driver) for SCO PC Unix comes on the CD. Please contact our support staff for more information on implementations.

5.8 Workstations

Our serial Hardlock SE is available for protecting software in workstations. Please contact our Sales Department for more information on using this Hardlock. The API does not support Hardlock SE.

6 High-level API Functions

In the following section we have provided you with a description of the various functions so that you can generate API calls in your particular high-level language. The exact procedure for implementing the individual functions in your programming language may differ slightly from those described here. All program examples are in pseudo code.

All functions are based on the API low-level functions. For most applications you will only need the high-level functions. However, we would be happy to send you a complete description of the low-level routines should you wish to use the programming interface of the low-level API. Excerpts from this documentation are also available as online text in API debugger TESTAPI.EXE.

In order to meet certain configuration requirements, the high-level API also emulates the functions of the old Hardlock implementations (before API was introduced). For more information on this subject, see Section 7.5.

On the following page you will find an explanation of how the individual high-level function descriptions are structured.

Structure of Function Descriptions

FunctionName ([Argument1[,Argument2[,...]]])

Purpose: Brief description of parameter

Arguments: List of arguments that can be used with this parameter.

Output: List of all values that this function outputs to the application (for example, error messages).

Use: Detailed description of how the parameter works, information on how it is used.

Example:

`The examples in this section are written in pseudo code and are only designed to roughly demonstrate how the function is used.`

6.1 HL_LOGIN (MOD, ACCESS, REFKEY, VERKEY)

Purpose: Initializes the API structure and defines the access mode. If applicable, it logs the application into the HL-Server.

Arguments:

MOD Hardlock module address.

ACCESS Used for defining the access mode:

1	LOCAL_DEVICE	Local Hardlock
2	NET_DEVICE	Access via the HL-Server
3	DONT_CARE	First searches locally, and then the network.

REFKEY If this string (8 bytes) is not equal to zero (binary), it is compared with the encrypted VERKEY value for identifying the Hardlock.

VERKEY (8 bytes) Contains the encrypted REFKEY value. The correct value for REFKEY (i.e. the VERKEY value decrypted with your Hardlock) can easily be determined with the API debugger program TESTAPI.EXE.

Output: The system outputs the API status (see "Table of API Status Values" in Appendix).

Use: This function provides the API structure with information on the Hardlock while initializing the API. If applicable, the application is logged into the HL-Server. All other accesses must occur between HL_LOGIN(...) and HL_LOGOUT(). This is why it is important to perform the HL_LOGIN(...) function at the start of an application. If DONT_CARE is entered, the API will first search for a local Hardlock and then look for one in the network.

The API automatically searches all parallel port addresses for the Hardlock. The default search sequence of the API is $378, $278 and then $3BC. Search of the serial ports can only be activated with the environment variable (also see Section 7.4.).

If you set the parameters REFKEY and VERKEY, they will automatically be used for cryptographically analyzing the Hardlock. (Note: the internal algorithm used for doing this is not compatible with K-EYE.)

To deactivate the REFKEY/VERKEY check, *binary zeros* must be used (not NULL as, for example, under C). Note: The REFKEY and VERKEY parameters in all demo programs only apply to the demo Hardlock (at module address 29809). To implement API for your own Hardlocks, you must generate a unique set of values with the TESTAPI.EXE program.

Example:

```
result = HL_LOGIN(29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
     .
 (Hardlock functions)
     .
  result = HL_LOGOUT();
ENDIF
```

6.2 HL_LMLOGIN (MOD, ACCESS, REFKEY, VERKEY, SLOT_ID, SEARCHSTR)

Purpose: Initializes the API structure and determines the access mode. If applicable, the application will log into the HL-Server. A check is run on the licensing information.

Arguments: **MOD, ACCESS, REFKEY, VERKEY**
See description of the function HL_LOGIN(...).

SLOT_ID Activates a check of the sublicensing information to a specific slot number. The range of values varies depending on whether the Hardlock is addressed on a local or remote system by HL-LiMa.

Local 1 to 728
Remote1 to 65534

SEARCHSTR Pointer on a Search String defined in the program. See section 7.4, Specifying the API Search Sequence.

Output: The return value contains the API status (see the "Table of API Status Values" in the appendix.

Use: This function is an enhanced HL LOGIN function (see HL LOGIN). A license check also takes place at a specific slot number. This function issues a sublicense to an application on the basis of modules and/or functions. HL-LiMa must be installed in order to issue licenses via a remote Hardlock..

It may be necessary to switch into several operating areas to run repeated Hardlock initializations in the same program. For more information, please refer to the description of the function HL SELECT(…).

You also have the option of defining the search sequence of the Hardlock API directly in the program. To do so, HL SEARCH must pass the parameters to the function according to the environment variables (for more information, refer to Section 7.4 Specifying the API Search Sequence).

Note: if the function (in sample C) is called with the parameters

```
HL_LMLOGIN (MODAD, ACCESS, REFKEY, VERKEY, 0, NULL)
```

it is completely compatible to the function HL_LOGIN(...).

Example:

```
result = HL_LMLOGIN(29809,REMOTE,"HARDLOCK","@0=/&#s3", 11,
"378p,IPX");
IF (result == TOO_MANY_USERS)
      PRINT "LICENCE PASSED LIMIT".
      HL_LOGOUT();
ENDIF
      .
      .
      .
```

6.3 HL_SELECT (DATA_AREA)

Purpose: Switches between the different work areas to use the Hardlock functions

Arguments: **DATA_AREA** Pointer on a data range of 256 Bytes.

Output: The return value contains the API Status (see the "Table of API Status Values" in appendix.

Use: It may be necessary to switch between several work areas for multiple initialization of Hardlock in the same program. To do so, data areas must be reserved in the program for managing the API structure. With the help of this function, you log into several Hardlock modules at the same time.
The function can also be used for sublicensing on a module or function basis. For example, a HL LMLOGIN(...) can take place at a different slot number for each application module.
A select in the area 0 sets the default value and uses the reserved internal data area only.

Example:

```
area1 = SPACE(256);
area2 = SPACE(256);

HL_SELECT(area1);
HL_LOGIN(29809,REMOTE,"HARDLOCK","@0=/&#s3");
HL_SELECT(area2);
HL_LOGIN(18328,REMOTE,"HARDLOCK","#:f;)?@0");
      ...
      ...
HL_SELECT(area1);
HL_LOGOUT();
HL_SELECT(area2);
HL_LOGOUT();
```

6.4 HL_LOGOUT()

Purpose: Releases the API structure. If applicable, the application is logged out of the HL-Server.

Arguments: (None)

Output: The system outputs the API status (See the "Table of API Status Values” in the Appendix).

Use: Use this function to release the API structure. If necessary, the system will first log out of the HL-Server. Once this function has been performed, the Hardlock can no longer be accessed. Therefore, it should only be called at the end of the application.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
      .
      .
 (Hardlock functions)
      .
      .
  result = HL_LOGOUT();
  QUIT;
ENDIF;
```

6.5 HL_PORTINF()

Purpose: Provides information on the port address of the Hardlock.

Arguments: (None)

Output: The system outputs the port address of the initialized Hardlock. In the case of an error, the value -1 is output.

Use: Use this function to determine the port address of the Hardlock initialized with the HL_LOGIN(...) function. It does not matter whether a local or remote Hardlock was found. This value should not be used for accessing through the HL-Server.

Example:

```
result = HL_LOGIN 29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
      .
      .
  Port = HL_PORTINF();
      .
      .
  result = HL_LOGOUT();
ENDIF;
```

6.6 HL_USERINF()

Purpose: Indicates how many entries are in the login table of the HL-Server.

Arguments: (None)

Output: The output value reveals how many entries have been made to the login table (including your own login entry) if a remote Hardlock was initialized. In the case of a local Hardlock, the output value is 1; in the case of an error, the output value is -1.

Use: This function indicates the number of users currently logged into the HL-Server. Use this function to issue licenses to access the network. Since the network can handle several protection Hardlocks with the same module address but different HL-Servers, API groups together all entries of the same Hardlock.

Example:

```
result = HL_LOGIN (29809,NET_DEVICE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_USERINF() > 5)
    PRINT "Too many users";
    result = HL_LOGOUT();
    QUIT;
  ENDIF;
      .
      .
  result = HL_LOGOUT();
ENDIF;
```

6.7 HL_AVAIL()

Purpose: Determines whether the Hardlock with the expected coding is available.

Arguments: (None)

Output: The system outputs the API status (see the "Table of API Status Values" in the Appendix).

Use: Use this function to determine whether a Hardlock with the expected coding is available (for example, before data is encrypted). If the values of REFKEY and VERKEY made available with the function HL_LOGIN(...) are not equal to zero, they are used for identifying the Hardlock. In the case of parallel access, the status of the select line of the printer port is also checked. When some printers are switched off, the lines are grounded and Hardlock cannot be recognized.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
      .
  IF (HL_AVAIL() == STATUS_OK)
      .
      .
 (Hardlock functions)
      .
      .
  ENDIF;
  HL_LOGOUT();
ENDIF;
```

6.8 HL_ACCINF()

Purpose: Determines whether a Hardlock was initialized as a local or remote Hardlock.

Arguments: (None)

Output: The following output values are possible:

1	LOCAL_DEVICE	The Hardlock was initialized as local
2	NET_DEVICE	The Hardlock was initialized as remote
-1	An error occurred.	

Use: If DONT_CARE was specified for the HL_LOGIN(...) function, HL_ACCINF() can be used to determine whether a local or remote Hardlock was initialized (found). The DONT_CARE value cannot be output after initialization.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_ACCINF() == NET_DEVICE)
    PRINT "Hardlock is remote";
  ELSE
    PRINT "Hardlock is local";
  ENDIF;
  result = HL_LOGOUT();
ENDIF;
```

6.9 HL_HLSVERS()

Purpose: Provides the version number of the HL-Server being used.

Arguments: (None)

Output: The system outputs the version number of the HL-Server being used as an integer (for example, 210 corresponds to version 2.10). This is <u>not</u> the API version of the HL-Server. In the case of a local Hardlock (or error) the value 0 is output.

Use: Use the function to determine the version number of the HL-Server being used. This information is important to have if a particular function is only available as of a certain version number.

Example:

```
result = HL_LOGIN (29809,NET_DEVICE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_HLSVERS() < 210)
    PRINT "Application needs HL-Server >= 2.10";
    result = HL_LOGOUT();
    QUIT;
  ENDIF;
      .
      .
  result = HL_LOGOUT();
ENDIF;
```

6.10 HL_CODE(DATAPTR, BCNT)

Purpose: Used for encrypting and decrypting data with the Hardlock.

Arguments:

DATAPTR	Pointer for the data area to be encrypted. This variable contains a 32-bit (far) real mode (segment:offset) pointer for a certain data area. You must use a 32-bit (near) protected mode (offset) pointer when working with the 32-bit API.
BCNT	Number of data blocks to be encrypted in 8-byte blocks. You can use a value between 0 (which wouldn’t make much sense) and 8192 (equivalent to 64 KB).

Output: The system outputs the API status (see the "Table of API Status Values" in the Appendix).

Use: This function encrypts and decrypts a given data area. The range of data specified is directly modified, i.e. it is directly overwritten by the result of the function. Calls with incorrectly set pointers thus often lead to undesirable results. In the case of an error, the data area specified can be destroyed. Since a block cipher is performed, the number of data blocks to be encrypted (64 bits each) must be specified. If you wish to encrypt data with a length of less than 8 bytes, fill up the range to the 8-byte limit. The application must ensure that the entire data block does not go beyond the segment limit (=pointers must be normalized) to avoid a wrap around.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  text = "Hello Hardlock !";
  IF (HL_CODE(text, 2) == STATUS_OK))
    PRINT "The encrypted data is: " + text;
  ENDIF;
  result = HL_LOGOUT();
ENDIF;
```

6.11 HL_READ(REG, VALUE)

Purpose: Reads the contents of a Hardlock memory register.

Arguments:

REG	Number of the register to be read. The register number must be between 0 and 63.
VALUE	Variable to be assigned the 16-bit register value and conveyed by reference.

Output: The system outputs the status of the Hardlock. If an invalid register is entered, the system issues the INVALID_PARAM error message (see the "Table of API Status Values" in the Appendix).

Use: This function is used to read an individual Hardlock memory register, that is, of course, if Hardlock is equipped with the Memory Option. The contents of the register is output as a 16-bit value (Intel format) in the VALUE variable.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_AVAIL() == STATUS_OK)
    value = 0;
    result = HL_READ (48, value);
  ENDIF;
    .
    .
  result = HL_LOGOUT();
ENDIF;
```

6.12 HL_READBL(DATAPTR)

Purpose: Reads the entire Hardlock memory to a certain data area.

Arguments: **DATAPTR** Pointer for the data area.

Output: The system outputs the API status (see the "Table of API Status Values" in the Appendix).

Use: Use this function to read the entire memory (registers 0 to 63). The data area defined with pointer DATAPTR must be 128 bytes in length.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_AVAIL() == STATUS_OK)
    eeprom = SPACE(128);
    HL_READBL (eeprom);
  ENDIF;
  result = HL_LOGOUT();
ENDIF;
```

6.13 HL_WRITEBL(DATAPTR)

Purpose: Writes the transferred data area to the RAM of the Hardlock.

Arguments: **DATAPTR** Pointer for the data area.

Output: The system outputs the API status (see the "Table of API Status Values" in the Appendix).

Use: Use this function to write to the entire ➔RAM of the Hardlock (registers 48 to 63). The data area must be 32 bytes in length. The contents of the RAM is maintained even when the Hardlock is disconnected (EEPROM).

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_AVAIL() == STATUS_OK)
    eeprom = SPACE(32);
    HL_WRITEBL (eeprom);
  ENDIF;
  result = HL_LOGOUT();
ENDIF;
```

6.14 HL_WRITE(REG, VALUE)

Purpose: Writes to one register of the Hardlock RAM.

Arguments:

REG	Number of the register to be written to.
VALUE	Value to be written to the register (16-bit value).

Output: The system outputs the API status (see the "Table of API Status Values" in the Appendix).

Use: Use this function to write to a single register of the Hardlock ➔RAM. The number of the register must lie between 48 and 63 of the RAM. The contents of the RAM are maintained even when the Hardlock is disconnected (EEPROM).

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  result = HL_WRITE(48, 147);
    .
    .
  result = HL_LOGOUT();
ENDIF;
```

6.15 HL_VERSION()

Purpose: Determines the version number of the API routines.

Arguments: (None)

Output: The API version number is output as an integer (for example, 350 corresponds to version 3.50). In the case of an error, the value -1 is output.

Use: Use this function to determine the version number of the local API being used. This is particularly important when working with an HL-Server since only API versions with the same main number are compatible. For example, an HL-Server with API version 2.xx is not compatible with an application which was manually implemented using API version 3.xx. Mixing different 3.xx versions, on the other hand, is perfectly acceptable. The API version of the active HL-Server cannot be determined via the network using a high-level API call. However, the VERSION_MISMATCH status code provides an indication of incompatible API versions

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  version = HL_VERSION();
  result = HL_LOGOUT();
ENDIF;
```

6.16 HL_MAXUSER()

Purpose: Determines the maximum (effective) permissible number of login entries.

Arguments: (None)

Output: Outputs the maximum permissible number of login entries, that is, if a remote Hardlock was initialized. In the case of a local Hardlock, the output value is always 1; in the case of an error, the output value is -1.

Use: When working with Hardlocks in a network, use this function to determine for how many "users" HL-Server is licensed. Since the network can handle several protection Hardlocks with the same module address but different HL-Servers, API groups together the entries of all Hardlocks with the same module address and coding.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_ACCINF() == NET_DEVICE)
     max = HL_MAXUSER();
     PRINT "HL-Server is licensed for: " + max;
  ENDIF;
ENDIF;
result = HL_LOGOUT();
```

6.17 HL_MEMINF()

Purpose: Determines whether the initialized Hardlock is equipped with the Memory Option.

Arguments: (None)

Output: The system outputs the API status (see the "Table of API Status Values" in the Appendix).

Use: Use this function to determine whether the initialized Hardlock is equipped with the Memory Option. If possible, the function tries to write and read a register. By comparing the results, the system determines whether a Hardlock with memory is present. Once the function has been performed the system returns to its original state.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  IF (HL_MEMINF() == STATUS_OK)
    PRINT "Hardlock with memory found.";
  ENDIF;
  result = HL_LOGOUT();
ENDIF;
```

6.18 HL_ABORT()

Purpose: Releases the API structure.

Arguments: (None)

Output: (Undefined)

Use: This function should only be used in emergencies (for example, within a critical error handler). All retrieved interrupts are restored. The API cannot perform any other functions once this function has been called. No new initializations may be performed. The application must be terminated as soon as possible.

Example:

```
   .
   .
HL_ABORT();
QUIT;
```

7 Appendix

7.1 API Debugger TESTAPI.EXE

Use the interactive API debugger program TESTAPI.EXE to test API functions and determine the API's behavior when accessing local Hardlocks and remote Hardlocks through the HL-Server without having to do any programming. The program can also be used for testing your Hardlock. It is self-explanatory and contains major portions of the API documentation as online text.

☞ Note:

Use the Latteccino program in the new Hardlock Bistro software to test the API functions. The software runs under Windows 95 or Windows NT only. For more information, please refer to the Hardlock Bistro manual.

The Latteccino program is an intuitive interface to the API. You only need to refer to this manual if you want to find out more about the individual API function.

7.2 Overview of High-level API Functions

The chart on the following pages provides you with an overview of all high-level API functions.

High-level Function	Use
HL_LOGIN (MOD, ACCESS, REFKEY, VERKEY)	Initializes the API structure and defines the access mode. If necessary, the applications is logged into the HL-Server.

High-level Function	Use
HLM_LOGIN (MOD, ACCESS, REFKEY, VERKEY, SLOT_ID, SEARCH_STR)	Initializes the API structure and defines the access mode. If necessary, the application is logged into the HL-Server or HLS-LiMa
HL_SELECT (DATA_AREA)	Toggles between the different work areas using Hardlock functions
HL_LOGOUT()	Releases the API structure. If necessary, the application is logged out of the HL-Server.
HL_PORTINF()	Outputs the port address of the Hardlock.
HL_USERINF()	Indicates how many entries are in the login table of HL-Server.
HL_HLSVERS()	Outputs the version number of the HL-Server being used.
HL_AVAIL()	Determines whether the Hardlock with the expected coding is available.
HL_ACCINF()	Determines whether the Hardlock was initialized as local or remote.
HL_CODE (DATA_POINTER, BCNT)	Encrypts or decrypts data with the Hardlock
HL_READ (REGISTER, VALUE)	Reads the contents of a Hardlock memory register.
HL_READBL (DATAPTR)	Reads the entire memory of the Hardlock with Memory into the specified data area.
HL_WRITEBL (DATAPTR)	Writes the transferred data area to the RAM of the Hardlock.
HL_WRITE (REGISTER, VALUE)	Writes to one register of the Hardlock RAM.
HL_VERSION()	Outputs the version number of local API routines.
HL_MAXUSER()	Outputs the maximum (effective) number of login entries possible.
HL_MEMINF()	Determines whether the initialized Hardlock is equipped with the Memory Option.

High-level Function	Use
void HL_ABORT()	Releases the API structure. This function should only be used in emergencies (for example, within a critical error handler).

7.3 Table of API Status Values

No.	Designation	Meaning
0	STATUS_OK	Function was performed without any errors.
1	NOT_INIT	API structure not initialized. The function cannot be executed. A successful HL_LOGIN(...) must first be performed.
2	ALREADY_INIT	The API structure is already initialized. A second attempt was made to initialize the API structure without it being released beforehand.
3	UNKNOWN_MODULE	Unknown module ID. This ID is not supported by the API. This problem is usually due to a programming error (incorrect pointer).
4	UNKNOWN_FUNC	Unknown API function number. An incorrect function number was used. This problem is usually due to a programming error (incorrect pointer).
5	(reserved)	
6	(reserved)	
7	NO_MODULE	No Hardlock connected. When working with a network, this problem can also be due to the following reasons: timeout, the station with the HL-Server cannot be addressed over the network.
8	NETWORK_ERROR	Error in network operation (remote). This may occur, for example, if no protocol (such as IPX or NetBios) is loaded in the local client system.
9	NO_ACCESS	The specified access mode is invalid (for example ACCESS = 0). This problem is usually due to a programming error (incorrect pointer).
10	INVALID_PARAM	An incorrect or invalid parameter was used with a function.

No.	Designation	Meaning
11	VERSION_MISMATCH	During communication, the system determined that the HL-Server or an installed Hardlock driver is not compatible with the API version being used.
12	DOS_ALLOC_ERROR	An error was detected in a routine that attempts to allocate memory space.
13	(reserved)	
14	CANNOT_OPEN_DRIVER	A required driver cannot be opened. The problem is usually that the driver is not installed.
15	INVALID_ENV	The specified environment variable for the API search sequence contains only incorrect entries.
17	INVALID_LIC	No valid licensing information found (e.g. local false memory or checksum error)
18	NO_LICENSE	Slot/license not released
256	TOO_MANY_USERS	This error code is issued when a station tries to login , but the HL-Server(s) is/are out of space in the login table.
257	SELECT_DOWN	If the Hardlock is not supplied with an operating voltage, it cannot operate and is thus no longer detected. The API is able to recognize this situation by measuring the Hardlock voltage level at the select line of the printer port. If this port is at ground potential, this is an indication that the other lines (from which the Hardlock gets its operating voltage) are also short circuited. For example, this occurs when certain printers are offline or switched off. The programmer can issue a message to the user, instructing him to either turn on the printer, switch it online or remove it from the interface. This error only applies to local operation.

7.4 Specifying the API Search Sequence

7.4.1 Background Information

It is possible to explicitly specify a search sequence with Hardlock API version 3.25 and higher. This is done through environment variables. By defining a specific search sequence, conflicts when the system automatically searches the LPT port addresses can be avoided (e.g. with network cards configured for LPT addresses).

An API search for Hardlock at the serial interface can **only** be activated through the environment variable.

7.4.2 Syntax

The syntax of the environment variables reads as follows:

```
HL_SEARCH=[Port],...,[Protocol],...
```

[Port] comprises the I/O address in hexadecimal form and a port ID:

Port ID:	Explanation:
p = parallel	Normal parallel port
s = serial	Normal serial port
e = ECP	Parallel port in ECP mode
n = NEC (Japan)	Japanese NEC models have a different port assignment. This parameter activates a special handling so that a separate NEC API is not required.
c = Compaq Contura Docking Base	The multiplexer of the docking base (used for switching between the parallel port and Ethernet adapter) is switched to the parallel port for querying a Hardlock.
i = IBM PS/2	The IBM PS/2 ID eliminates errors when reprogramming the port of certain video drivers under Windows. (The system cannot find Hardlock once Windows is started). Previously, this was done internally by the Hardlock API. This function can now only be activated by using the environment variable.

[Protocol] defines the protocol used for accessing an HL-Server. The following key words are currently supported:

Protocol:	Explanation:
IPX	HL-Server searched for via IPX or SAP.
IP	HL-Server searched for via TCP/IP
NETBIOS	HL-Server searched for via NETBIOS.

Example:

```
SET HL_SEARCH=378p
```

The system only searches for the Hardlock at the local parallel port with the address 0x378.

```
SET HL_SEARCH=378e,2f8s
```

The system searches for the Hardlock at the local parallel port with the address 0x378. The port is switched from ECP mode to "normal” mode while the Hardlock is being accessed. If the system cannot find the Hardlock, it then searches for it at the serial port with the address 0x2f8.

```
SET HL_SEARCH=IPX,278p
```

The system first searches for a Hardlock supplied by the HL-Server using IPX/SAP. If it is not able to log into the HL-Server, it then searches for the Hardlock at the local parallel port with address 0x278.

```
SET HL_SEARCH=378p,278p,3BCp,IPX,NETBIOS, IP
```

This corresponds to the automatic search sequence (HL_LOGIN with DONT_CARE) if the environment variable is not defined. This entry is thus redundant.

The search sequence can be directly specified through direct programming of the low-level API.

7.4.3 HL-Server Client for TCP/IP

When using the 32-bit HL-Server for Win95 and Windows NT please bear in mind that the search sequence of the protocols (if you're not using HL Search) depends on the client. Thus:

16 Bit Search Sequence: IPX, NetBios, IP

IP is searched last in order to change the former behavior as little as possible.

32 Bit Search Sequence: IP, IPX

IP is searched first since this search is considerably faster than via IPX or NetBios.

Search Sequence for IP Addresses:

- Environment variable HLS_IPADDR (see below)
- If no environment variable has been defined, the search takes place via DNS or HOSTS for the station HLSERVER.
- If no address has been found, the search takes place via broadcast (255.255.255.255) in the local segment.

To transfer IP packets, Winsock calls over a corresponding (16 or 32 bit) WINSOCK.DLL. Please note that during installation many Internet clients install their own WINSOCK.DLL (CompuServe, AOL, T-online). In this case, accessing IP calls the Internet provider if the HL-Server was not found via IPX and NetBios. You then need to exclude IP from the search with the following command:

```
SET HL_SEARCH=IPX,NetBios
```

In order to improve the search via the TCP/IP protocol, the environment variable HLS IPADDR has been introduced. With it one or more IP addresses or names can be defined. However, by entering several addresses at the same time - in contrast to HL SEARCH - you cannot predict which of the defined HL-Servers will ultimately be used.

Example:

```
SET HLS_IPADDR=192.9.209.17,luzie.fast.de
```

HLS IPADDR can also be used to define broadcast addresses:

```
set HLS_IPADDR=192.9.209.255,192.9.201.255
```

Since IP networks generally have considerably greater differences in propagation time than IPX networks (e.g. WAN routes), timeouts and retries for the clients must be kept within bounds. The default values are set so that the HL-Server can be found with an existing 64kbit connection.

SET HLS_WAIT=
sets the delay between retries in milliseconds

default	TCP/IP:	1000,
	IPX: 200	(*)
min	200	
max	30000	

SET HLS_RETRIES=
sets the number of retries until message DONGLE_NOT_FOUND is returned

default	5
min	2
max	30

(*) the defaults vary, SET HLS_WAIT changes the values for IPX and TCP/IP !

7.4.4 Search Strategy

Please keep the following points in mind when specifying the access type with HL_LOGIN:

- HL_LOGIN(MODAD, LOCAL_DEVICE,....)

The system searches all local ports without the environment variable. By specifying the environment variable you can direct the system to only search local addresses (parallel and serial). It is not possible to subsequently instruct the system to search the network.

With:

```
SET HL_SEARCH=IPX,278p
```

only the address 0x278 is used. IPX is ignored.

With:

```
SET HL_SEARCH=IPX
```

a Hardlock will not be found since the entry is overruled by the access type specified with HL_LOGIN. HL_LOGIN issues error code 15 (INVALID_ENV).

- HL_LOGIN(MODAD, NET_DEVICE,....)

Here the system searches all supported protocols for an appropriate HL-Server without an environment variable. By specifying an environment variable, you can only restrict the protocols used for searching. It is not possible to subsequently instruct the system to search the local ports.

With:

```
SET HL_SEARCH=IPX,2f8s
```

only the IPX protocol is used. 2f8s is ignored.

With:

```
SET HL_SEARCH=278p
```

a Hardlock will not be found since the entry is overruled by the access type specified with HL_LOGIN. HL_LOGIN issues error code 15 (INVALID_ENV).

- HL_LOGIN(MODAD, DONT_CARE,....)

Without the environment variable, the system first searches all local parallel ports. It then searches for an appropriate HL-Server with all supported protocols. By specifying the environment variable you can restrict the search in any way you like.

7.4.5 Comments

- If the environment variable does not contain any valid entry, the HL_LOGIN function issues error code 15 (INVALID_ENV).
- It does not matter whether the environment variable is in small or capital letters.
- When working with Windows programs, the environment variable must be specified before Windows is booted. Subsequent modification in a DOS box has no effect on Windows programs.
- Specifying a port address ensures that your specific Hardlock is supported. For example, with "SET HL_SEARCH=320p" the system searches for the Hardlock at port address 0x320. Entering an incorrect port address can lead to conflicts.
- Programs encrypted with HL-Crypt (version 5.64 or higher), HLWCrypt (version 4.06 or higher) and HLCWin32 (version 1.03 or higher) search for a Hardlock according to the rules outlined above.

- The search sequence of serial ports is only supported by API version 3.50 and higher.

7.5 Compatible Calling Conventions

The functions

- HL_ON (port address, module address)
- HL_OFF (port address)
- HL_RD (port address, register)
- HL_WR (port address, register, value)
- INT_ON ()
- INT_OFF ()
- K_EYE (port, data pointer, block count)

of the old implementations (before the API) have been re-implemented due to compatibility reasons. The API libraries let you use the expanded API capabilities. Only implementation with access to local Hardlocks is possible.

Structure the commands as shown in the following example:

```
INT_OFF()
  HL_ON(...)
    HL_WR(...)
    HL_RD(...)
    K_EYE(...)
  HL_OFF(...)
INT_ON()
```

They are now directly linked to the API; various parameters and functions are ignored since these are automatically handled by the low-level API.

If you are developing new software or expanding existing systems and wish to take advantage of all API capabilities, do not use these functions. These functions were only implemented for exceptional cases in which compatibility is of utmost importance. Do not mix new API functions with these old functions.

☞ Note:

When programming with the new Hardlock API, please keep in mind that the functions HL_LOGIN(...) and HL_LOGOUT() ***are not*** the same as HL_ON(), and HL_OFF() since actual activation of the Hardlock occurs automatically within the API.

7.5.1 Hardlock Module in Compatible Mode

Parallel calculation was also implemented in the API for Hardlocks that operate in the old parallel calculation mode. The corresponding function expects 4 times 8 bits as 32-bit argument and delivers an 8-bit value.

Example:

```
result = HL_LOGIN (29809,DONT_CARE,"HARDLOCK","@0=/&#s3");
IF (result == STATUS_OK)
  value = HL_CALC (i1, i2, i3, i4);
ENDIF;
result = HL_LOGOUT();
```

7.6 Naming Conventions of API Objects

To simplify and better identify the different libraries, we have developed a system for naming the API files. The file name is made up of 4 characters "**API_**" and a 4 digit code. The meaning of the different digits of the code are explained in the table below. The file extension is **OBJ** or **LIB** for libraries which correspond to the Microsoft **OBJ/LIB** standard or **DLL** if they are dynamically loadable libraries.

Library name:	**API**	_	**x**	**x**	**x**	**x.ext**
	16-bit API	=	**1**	**.**	**.**	**.**
	32-bit API	=	**3**	**.**	**.**	**.**
	DOS generic OBJ	=	**.**	**D**	**.**	**.**
	DOS only code OBJ	=	**.**	**B**	**.**	**.**
	Win 3.xx generic OBJ	=	**.**	**W**	**.**	**.**
	Win 3.xx generic DLL	=	**.**	**L**	**.**	**.**
	Win 3.xx Watcom OBJ	=	**.**	**X**	**.**	**.**
	Win NT/Win32 OBJ	=	**.**	**N**	**.**	**.**
	OS/2 DLL	=	**.**	**M**	**.**	**.**
	PC UNIX	=	**.**	**U**	**.**	**.**
	Novell Netware NLM OBJ	=	**.**	**S**	**.**	**.**
	DOS DPMI extender OBJ	=	**.**	**P**	**.**	**.**
	NET/REMOTE + local routines	=	**.**	**.**	**N**	**.**
	local only	=	**.**	**.**	**L**	**.**
	no multitasking support	=	**.**	**.**	**.**	**N**
	multitasking shell	=	**.**	**.**	**.**	**M**