
# NOVOS

*ver. 3.1*

**N***y-*

**O***perativsystem*

**V***irtuelledisk*

**O***peration*

**S***imulator*

# TECHNICAL MANUAL

*English Version*

# TABLE OF CONTENTS

# INTRODUCTION

## What is NOVOS?

NOVOS (*NyOperativsystem Virtuelledisk Operation Simulator*) is a set of two low level kernel simulation and terminal-based virtual disk/file management programs based on the NO (*NyOperativsystem*) operating system developed by Havland Technica in 1999. The final version of NO has been released in 2010 as v7.1.2, and NO will reach EOL (end-of-life) in 2021 as we cease to provide security and content updates. As we are aware that many legacy users of our products continue to utilise computers with NO proprietary format encrypted hard disks, NOVOS has been developed to ease the transition from NO-based computers to computers with modern operating systems (Windows/Mac OS X/Linux).

## Using NOVOS

Your NOVOS product is available either via software download or as a standalone disk. It is intended for use on a modern personal computer or laptop. Specifications for NOVOS are as follows:

- **OS:** Windows 7 or above/Mac OS X Snow Leopard or above (For Linux distributions please contact customer support)
- **Memory:** 1GB RAM or above, 200GB secondary storage or above (Depending on NO-format hard disk size)
- **Processor:** Pentium Dual-Core E2100 series or above

There are two components: `NOVOS-CONV` and `NOVOS-RW`. Both are licensed to your personal account details which you provided on purchase, and will not work on computers with different usernames and passwords.

Once installed, `NOVOS-CONV` will automatically seek out any connected NO-format hard drives with your credentials, read from the hard drive memory, and convert the contents of the hard drive into an encrypted `.vd` (*virtuelledisk*) format suitable for use and storage on modern computers. `NOVOS-CONV` supports all NO-format hard disks from versions 1 to 4.

Upon completion, `NOVOS-RW` will allow you to read and preview files on the `.vd` drive and transfer files into a format suitable for your operating system. To achieve this `NOVOS-RW` will simulate the NO operating system and act as a bridge to convert files to their destination equivalent formats. Depending on hard disk version, `NOVOS-RW` will also apply the appropriate decryption scheme (DES for version 1, AES-256 for versions 2-4) using your credentials.

# NOVOS-CONV

## Overview

As the NO-format hard disk remains proprietary technology, we are unable to offer information about the operation of `NOVOS-CONV`. `NOVOS-CONV` does not require user credentials to function, as it does not perform any interpretation of file data or decryption of files.

## Operation

Decompress `NOVOS-CONV` and navigate to the folder containing `NOVOS-CONV` using a terminal or command prompt. Then enter the following command.

- **Windows:** `novos_conv.exe`
- **Mac OS X/Linux:** `./novos_conv`

## Error Code Listing

In the event that the program produces error codes, please see the following reference.

- 0x12 - Physical disk corruption *(Please check your hard disk is in working order)*
- 0x15 - Manual Interrupt *(Caused by user manually terminating process early)*
- 0x20 - Out of Memory *(Please increase hard disk memory available to target computer)*
- 0xFF - Unknown Error *(May be related to local system)*

For all other error codes or non-standard terminations, please contact customer service.

# NOVOS-RW

## Overview

NOVOS-RW is based on the architecture of a simulated NO computer, albeit one without many components including networking, external device support, video or audio drivers, data stream processing etc. However, it retains many of the user-friendly features of the NO operating system, including the manual debugging mode which many users find a preferrable alternative to waiting for costly repairs. We are also aware of the danger of handing a copy of software containing user credentials to a third party for repair or analysis. Therefore this manual will also cover non-standard operation of NOVOS-RW, particularly in the event of a corrupted or non-functioning copy of NOVOS-RW.

## Operation

Decompress NOVOS-RW and navigate to the folder containing NOVOS-RW using a terminal or command prompt. Then enter the following command, replacing {CUSTOMER_ID} with your Havland Technica customer ID number.

- **Windows:** `novos_rw_{CUSTOMER_ID}.exe`
- **Mac OS X/Linux:** `./novos_rw_{CUSTOMER_ID}`

## Error Code Listing

In the event that the program produces error codes, please see the following reference.

- 0x12 - Physical disk corruption *(Please check your hard disk is in working order)*
- 0x15 - Manual Interrupt *(Caused by user manually terminating process early)*
- 0x20 - Out of Memory *(Please increase hard disk memory available to target computer)*
- 0x22 - Credentials Incorrect *(Please ensure correct credentials were supplied)*
- 0xFF - Unknown Error *(May be related to local system)*

For error codes in the 0x31-0x3F series, the system will enter/utilise manual debug mode (see below).

## Manual Debug Mode Overview

In the event that the program displays a message such as "*Starting in manual debug mode (Use command "help" for help)...*", you have entered manual debug mode. You should then see the prompt "`(dbg)`".

If you encounter manual debug mode, this means that the system believes it has isolated an assembly-based syscall or subroutine that isn't behaving correctly. When you correct the code to a correct state (as demonstrated by it producing the target memory state), the system will continue to the next malfunctioning syscall, until all such errors are manually resolved. See **Appendix I** for how to interpret NO-format assembly code.

## Manual Debug Mode Command Listing

See below for a list of commands available in manual debug mode:

- `name`: Gives the name of the syscall. *Example: "name" will return "Swizzle Function" for a syscall that implements the swizzle function.*
- `tgt`: Allows you to view the target memory state of the virtual memory allocated to this syscall. Your aim is to make sure that at the end of program execution memory resembles the output of `tgt`. *Example: "tgt" shows you the relevant memory locations for the current syscall. (? means any value is acceptable at that memory location)*
- `mem`: Allows you to inspect the current state of the virtual memory allocated to this syscall. *Example: "mem" after a program has crashed allows you to examine the memory at the moment of the crash.*
- `diff x`: Gives you the memory limit for this patch (the number of characters you can add/edit/delete) and shows the difference between the starting line `x` and the current (patched) line `x`. If you haven't changed line `x`, there will be no difference. If `x` is not provided it diffs the entire source code. *Example: "diff 10" will show differences in line 11 of the source code (ALED is 0-indexed).*
- `res`: Resets memory to initial state. Useful for restarting programs. Patched code is NOT reset. *Example: "res" after a crash resets memory to the original state.*
- `marks`: Prints all marks. Marks are points code can jump to using JIF or JMP. *Example: "marks" in a code with 1 mark will print that mark and the line number associated.*
- `print x`: Prints the current state of the code at line `x`. If `x` is not provided it will print the entire code block with line numbers. *Example: "print 10" prints the code at line 11.*
- `orig x`: Prints the original state of the code at line `x`. If `x` is not provided it will print the entire code block. *Example: "orig 10" prints the original unpatched code at line 11.*
- `patch x newline`: Replaces line `x` with the contents of `newline`. Shows you how many characters have changed. *Example: "patch 10 SUB $0 $1 $2" will replace the contents of line 11 with "SUB $0 $1 $2".*

- `run x`: Runs `x` lines of code, then stops until you invoke run again. If `x` is not provided will run until it errors or completes. *Example: "run 2" will run the next 2 lines of code.*

# APPENDIX I: NyOperativsystem 64-bit Assembly (NO64-ASM) Syscall Specification

> **WARNING: This information can only be used to debug and patch NOVOS or other NO-format assembly based products in manual debug mode. Creating and/or distributing unlicensed operating system modifications for Havland Technica products voids any available warranty and may violate the terms of use.**

## Overview

A NO64-ASM syscall is a process spawned by NO, usually to complete a specific operation whether mathematical (e.g. calculating some vector transformation) or internal (e.g. memory manipulation). A basic syscall process will consist of the following:

- One page of allocated virtual memory (100 memory locations each capable of storing a 64-bit signed float, labelled $0, $1, $2 etc. up to $99)
- An assembly program used to achieve the syscall (read-only, a syscall program can never alter itself)
- An instruction pointer ($IP) register used to control program flow
- An initial and a target memory state (read-only, a syscall program can never alter its target memory state) used to verify the syscall on startup

## Instruction Set

An *instruction* consists of a *command* and a number of *arguments*. `ADD $0 1 $99` has the command `ADD` and the arguments `$0`, *1*, and `$99`. Arguments can be expressed as either integers (*1, 2, -15*) or memory locations (`$1`, `$3`, `$99`). However, some arguments only accept memory locations (usually those specifying where to write the result of an operation to). Some arguments also accept marks such as `:loop`.

### Arithmetic

- `ADD val1 val2 dest`: Adds the values at memory locations `val1` and `val2` and writes the result to memory location dest. `ADD $0 $1 $2` adds the values at `$0` and `$1` and writes the output to memory location `$2`.
- `SUB val1 val2 dest`: Subtracts the values at memory locations `val1` and `val2` and writes the result to memory location `dest`.

### Testing

- `TEQ val1 val2 dest`: Tests if the values at memory locations `val1` and `val2` fulfill the condition `val1 == val2`. If yes, writes *1* to memory location `dest`. If not, writes *-1*.
- `TLT val1 val2 dest`: Tests if the values at memory locations `val1` and `val2` fulfill the condition `val1 < val2`. If yes, writes *1* to memory location `dest`. If not, writes *-1*.
- `TGT val1 val2 dest`: Tests if the values at memory locations `val1` and `val2` fulfill the condition `val1 > val2`. If yes, writes *1* to memory location `dest`. If not, writes *-1*.

**Flow Control**

- `MRK name`: Creates a mark with the name `name`. `MRK START` creates a mark with the name `START` that can be referenced with `:START`.
- `JMP :name`: Jumps to the mark with the name `name` and continues executing.
- `JIF loc tgt`: If the value at memory location `loc` is positive (i.e. `loc >= 1`) jump to the location specified in `tgt`. This can be the value of another memory location or a mark. `JIF $1 $2` means if the value at `$1` is positive, jump to the line of code with line number specified by `$2`. `JIF $1 :START` means if the value at `$1` is positive, jump to the mark with the name `START`.

**Memory read/write**

- `MOV loc dest`: Copies the value from the memory location `loc` to the memory location `dest`. `MOV $0 $1` means that `$1` will now have the value of `$0`.
- `MRD loc dest`: Reads the memory location specified by the value of `loc` and copies it to the memory location `dest`. `MRD $0 $1` where `$0` has a value of *5* means that the system will copy the value at memory location `$5` to `$1`.
- `MWT loc dest`: Writes the value at memory location `loc` to the memory location specified by the value of `dest`. `MWT $0 $1` where `$1` has a value of *5* means that the system will copy the value at memory location `$0` to `$5`.