

# NICOF

## *Non-Invasive* **CO**mmunication *F*acility

Version 0.6.0 for VM/370 R6 SixPack 1.2

Dr. Hans-Walter Latz, 30.07.2014

Berlin, Germany

### **WARNINGS:**

This software is delivered as-is with no promise or commitment to be usable for any particular purpose.

Use it at your own risks!

The NICOF software and documentation has been written by a hobbyist for hobbyists and should not be used for any important or even critical tasks.

NICOF is work in progress and its current implementation may differ from this document.

### **License:**

NICOF and the NICOF sources are released to the public domain.

**To all those working in or for the three- or four-letter agencies and their governments who still believe that they are allowed to violate other peoples right for free and undisclosed communication just because they can do it at the world-wide scale, putting free speech and free press at stake:**

No, you don't have this right!  
STOP WATCHING US!

George Orwell's "1984" was not meant to be an instruction manual

*(unknown)*

Als sie die Kommunisten holten, habe ich geschwiegen; ich war ja kein Kommunist.  
Als sie die Sozialdemokraten einsperrten, habe ich geschwiegen; ich war ja kein Sozialdemokrat.  
Als sie die Gewerkschafter holten, habe ich nicht protestiert; ich war ja kein Gewerkschafter.  
Als sie die Juden holten, habe ich nicht protestiert; ich war ja kein Jude.  
Als sie mich holten, gab es keinen mehr, der protestieren konnte.

When they came for the communists, I remained silent; I was not a communist.  
When they locked up the social democrats, I remained silent; I was not a social democrat.  
When they came for the trade unionists, I did not speak out; I was not a trade unionist.  
When they came for the Jews, I didn't speak up, because I wasn't a Jew.  
When they came for me, there was no-one left to speak out.

*Pastor Martin Niemöller*

Wehret den Anfängen!  
Nip things in the bud!  
Resist the beginnings

*(german saying)*  
*(as translated by [www.leo.org](http://www.leo.org))*  
*(my literal translation)*

## Contents

1	Introduction.....	5
1.1	Overview.....	5
1.2	How does it work? .....	6
1.3	Performance .....	9
2	Installing NICOF .....	11
2.1	Content of the delivery packages.....	11
2.2	Prerequisites.....	13
2.2.1	Java runtime .....	13
2.2.2	Hercules.....	13
2.3	Installation of the Playground package .....	13
2.4	Installation of the Standard package.....	15
2.4.1	Extending the Hercules configuration for VM/370 .....	15
2.4.2	Setup in VM/370.....	16
2.4.3	Starting the NICOF inside proxies.....	16
2.4.4	Accessing the NICOF tools .....	17
2.4.5	Starting the outside proxies .....	18
2.4.6	Configuration of the outside proxy .....	19
2.5	Installation of the Manual package .....	19
2.5.1	VM/370 internal configuration steps .....	19
2.5.2	External configurations steps .....	20
2.6	NICOF Source code .....	20
3	Using NHFS .....	22
3.1	Naming conventions and restrictions for outside files .....	22
3.2	Configuring the outside service.....	22
3.3	NHFS subcommands.....	22
4	Socket API for CMS and FTP server for CMS .....	24
4.1	Supported BSD-like socket API .....	24
4.2	CMSFTPD – FTP server for CMS.....	26
4.3	Socket API implementation .....	29
5	Doing the load tests.....	30
6	NICOF API documentation.....	31
6.1	Interrupt API for C .....	31
6.1.1	Type shortcuts .....	32

6.1.2	Initialisation .....	32
6.1.3	Handling external interrupts .....	32
6.1.4	Handling device (internal) interrupts .....	32
6.1.5	Posting / waiting for ECBs .....	33
6.1.6	Timer facility .....	34
6.1.7	VMCF interface .....	34
6.1.8	SIO support .....	35
6.2	Level-0 API .....	36
6.2.1	CMS client-side Level-0 API .....	36
6.2.2	Java server-side Level-0 API .....	40
6.3	Level-1 API .....	41
6.3.1	CMS Level-1 API .....	41
6.3.2	Java server-side Level-1 API .....	46

# 1 Introduction

## 1.1 Overview

NICOF (**N**on-**I**nvasive **C**ommunication **F**acility) gives VM/370 running on Hercules the ability to actively communicate with the outside world of the emulated mainframe.

At an abstract level, it allows *inside* processes (CMS programs running in VMs under VM/370) to exchange data packets with *outside* services implemented in a Java process. The terms *inside* and *outside* will (attempted to) be used consistently to identify the programs/processes *inside* VM/370 resp. the processes *outside* Hercules implementing the functionality communicated with.

The active part of the communication is in all cases the *inside* process (a client program under CMS), which initiates each communication by sending a request (data packet with metadata) to the *outside* service implementation, which processes the request and sends back a response (data packet with metadata) to the client.

From an end-user perspective, NICOF currently mainly provides 2 useful programs:

- **CMSFTPD**, a FTP server running in a VM under CMS, providing access to the minidisks of this VM from outside the VM/370 resp. Hercules machine with standard FTP clients.  
The FTP server supports the standard operations of the FTP protocol, i.e. downloading files from the VM to the FTP client as well as uploading, deleting and renaming files on writable CMS disks.
- **NHFS**, a CMS program allowing to access a user specific part of the file system on the outside of the VM/370 resp. Hercules machine. Accessing means:
  - list and create directories on the outside host
  - transfer files between the outside host and the CMS A-disk.Accessing files is intentionally restricted to files and directories following naming rules similar to the rules for CMS files. Furthermore, only files and directories in a configured exchange area (base directory) can be accessed with NHFS.

From a programmer's perspective, NICOF provides a set of APIs ranging from high-level to low-level:

- a socket API intended to be compatible with the socket API found on UNIXoids or Windows; NICOF currently provides TCP/IP stream sockets to CMS programs.  
This socket API does not implement a TCP/IP stack under VM/370, but “merely” provides access to the communication infrastructure of the underlying operating system where Hercules runs (more precisely where the Java process providing the outside proxies runs)
- a STDIO-like API allowing to access files on the system hosting the Hercules emulator (NHFS-API)
- several NICOF specific APIs allowing to implement high-level-APIs for communication services like the socket or NHFS APIs with the NICOF communication infrastructure, including a very low-level API giving access to typical assembler functionality to the C language (like SIO or VMCF invocation, interrupt handling, ...)

As the term “non-invasive” indicates, NICOF does not need modifications or extensions to VM/370 R6 or to Hercules. It works with the on-board internal and external communication means which are part of VM/370 resp. Hercules. However, a Java Runtime Environment is needed to provide the outside infrastructure and

services. Furthermore, a fix to Hercules is required for a Hercules version 3.x to avoid serious transfer speed restrictions (this fix is already included in Hercules 4.00 since January 2013).

The source code for the mentioned NICOF functionality is provided with the binaries, along with some test tools used during development.

## 1.2 How does it work?

In a nutshell:

- NICOF allows CMS programs executing in client-VMs on the inside to interact with services on the outside, these outside services are implemented in Java and allow accessing functionality of the underlying operating system (files, sockets etc.)
- in between these end-to-end communication partners is a gateway, consisting of a VM on the inside and a Java process on the outside
- the client programs access an outside service through a gateway with the NICOF client-API, which uses VMCF-requests to the inside VM of the gateway, or with a high-level API built on top of the NICOF client-API

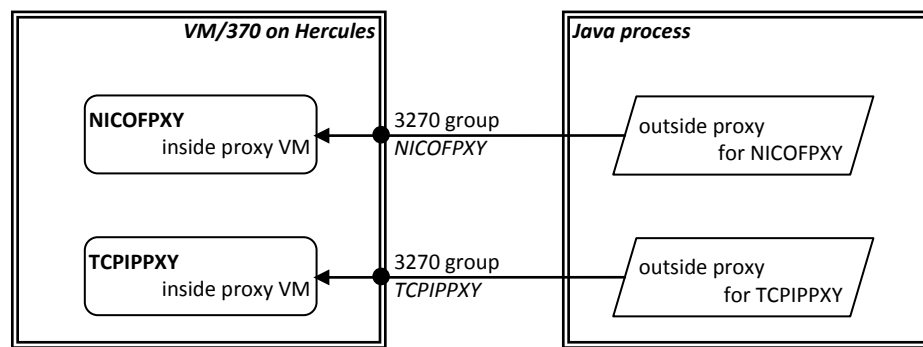
The NICOF architecture and concepts will be described in this section in terms of the setup necessary for support both the CMSFTPD and the NHFS program for end users. This is the setup delivered with the NICOF installation packages.

The basic inside-outside gateway is provided by a virtual machine on the inside, which exchanges data packets with an external process through a 3270 terminal line. The virtual machine is called the inside proxy and runs the NICOF proxy program (`IOPROXY MODULE`). The external process is a Java program running the outside proxy implementation (a Java class). The outside proxy is connected to VM/370 on Hercules through the 3270 console port and is DIALED to the inside proxy VM. The data packets are exchanged between the 2 proxies through screen writes (inside → outside) and simulated user inputs (outside → inside) initiated by the NICOF proxy program through SIO operations to the attached 3270 terminal simulated by the outside proxy.

When the Java process is started, one or more properties files are specified on the command line, each file defining one outside proxy to run in the process. Each outside proxy connects to the configured Hercules machine and attempts to DIAL to its inside proxy VM, retrying the connect-DIAL cycle after a while if a connect fails or a working connection is lost (e.g. the inside proxy VM crashed or was FORCED). The startup sequence of the inside and outside proxies is not important, as the outside proxy will retry until DIALing eventually succeeds and the gateway becomes functional. To allow for a reliable connecting procedure, the Hercules configuration for the VM/370 system must define a separate 3270 terminal group with exactly one device for each inside proxy, with the group having the (case-sensitive) name of the proxy VM: the outside proxy always connects to Hercules using the inside proxy VM name as LU-name for the 3270 line.

The standard NICOF setup uses 2 gateways, i.e. 2 inside proxy virtual machines. The user **NICOFPPXY** is intended to provide standard services like NHFS or some test APIs, whereas the user **TCPIPPXY** is dedicated to support the socket API and related functionality.

The following picture summarizes the low-level inside-outside communication paths in the standard setup:



In the standard setup, the 2 outside proxies are provided a single Java process running on the same host machine as Hercules. Alternatively, each outside proxy may be provided by a dedicated Java process, each of which could run “anywhere”, i.e. on the same computer as Hercules or a different machine (as long as the VM/370 system on Hercules can be reached).

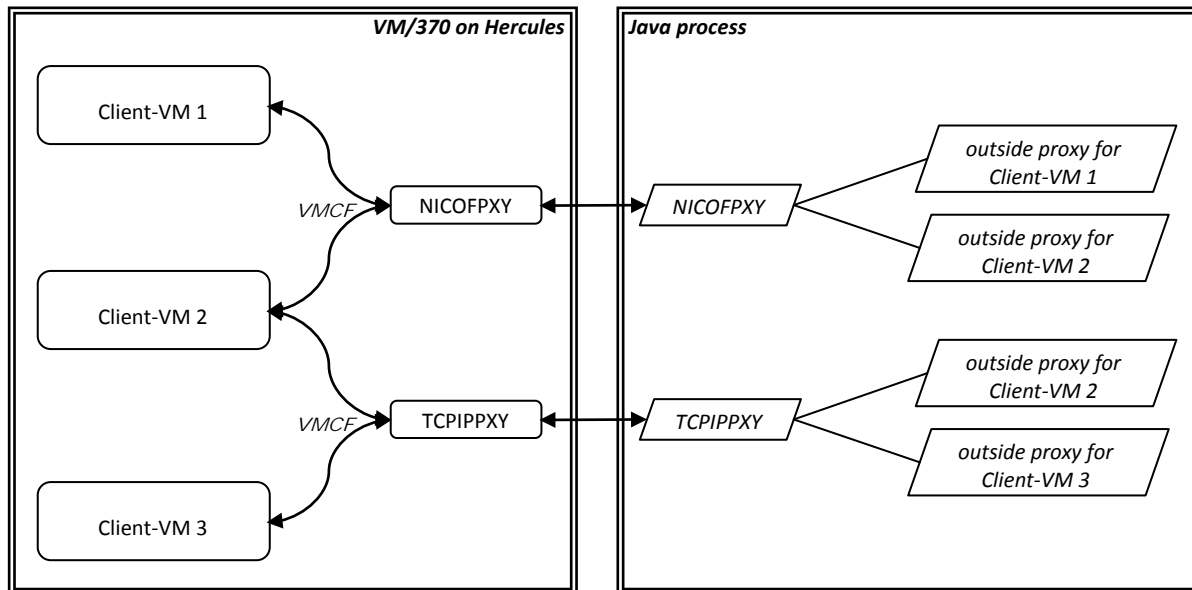
A gateway simply forwards data packets between the inside and the outside, i.e. neither the inside proxy VM nor the Java proxies are active by themselves. An active communication is always initiated by a client VM which invokes a service on the outside, with the following basic scenario:

- a request is prepared on the client VM and sent to the proxy VM using VMCF (Virtual Machine Communication facility of VM/370 R6);
- the inside proxy forwards the request packet to the outside proxy with additional return information, mainly the name of the client VM originating the request and an internal request identifier;
- on the first request from this client VM, a new outside proxy dedicated to this client VM is instantiated, this proxy instance will be used for all subsequent requests from the same client VM; the Java class to instantiate is configured in the properties file for the gateway;
- each request is processed asynchronously by the outside proxy instance, eventually sending the response along with the return information back to the inside proxy;
- the inside proxy transmits the response as VMCF reply to the VM which initiated the request.

Both the request and the response consist of a data packet with 0 to 2048 bytes and 2 fullwords as additional metadata sent along with the data packet (so called userwords at the discretion of the client-service protocol for sequencing requests, selecting subservices or functions, returncodes etc.). As requests are processed asynchronously by the outside proxy, more than one request from a given VM to the same outside service may be active at a time and the responses to requests may arrive in a different order than the request sequence. A gateway can of course be used simultaneously by several client VMs.

A single client VM may need to communicate with more than one gateway (inside proxy) to access the outside services needed by a client program. This is supported by NICOFP, the gateway to use for a given outside service (i.e. which inside proxy VM to address) depends of course on which outside proxy is configured to work with the particular Java class implementing the service, but this detail is typically hidden to the client program by the high-level API providing access to the service.

The following picture summarizes the communication paths and the instantiated outside proxies for 3 client VMs , with one accessing 2 outside services:



The NICO communication APIS support 2 abstraction levels regarding service differentiation:

- The lowest level is the request-response mechanism described above with a data packet (0..2048 bytes) and 2 fullwords for metadata. This API level is protocol and service agnostic and is called **Level-0**. The Java class for the outside proxy will receive and process the Level-0 requests. The implementation of the class must of course match the expectations of the client programs sending the requests regarding the meaning of the packets and userwords sent forth and back as well as the functionality requested.
- A specific Level-0 service is provided with NICO which allows to provide several services in parallel through a single outside proxy (and therefore a single gateway), using a unified programming model both on the inside and the outside. This **Level-1** service infrastructure reserves one of the userwords to transport the information about the particular service to be invoked (service-id) and the function to be called on that service for the request (command) and the return code for the response. In the properties file the outside proxy, a list of Level-1 service classes with their associated service name is specified. For each client-VM issuing requests to the outside, a separate set of instances of these service implementations is instantiated when the proxy instance for the client VM is created on the first request. A name service is automatically added to the Level-1 services, allowing to resolve a service name to a service-id for the configured services.  
On the inside, the API for Level-1 infrastructure allows to resolve a service name to a service-id and to invoke a command on a service, passing a data packet and a control word (the last remaining userword) and receiving a data packet along with a returncode and a controlword. Furthermore, the Level-1 infrastructure supports STDIO-like streams for data exchange between the inside and outside.  
On the outside, the service proxy dispatches the invoked command to the service instance identified by the service-id and returns the response to the inside.

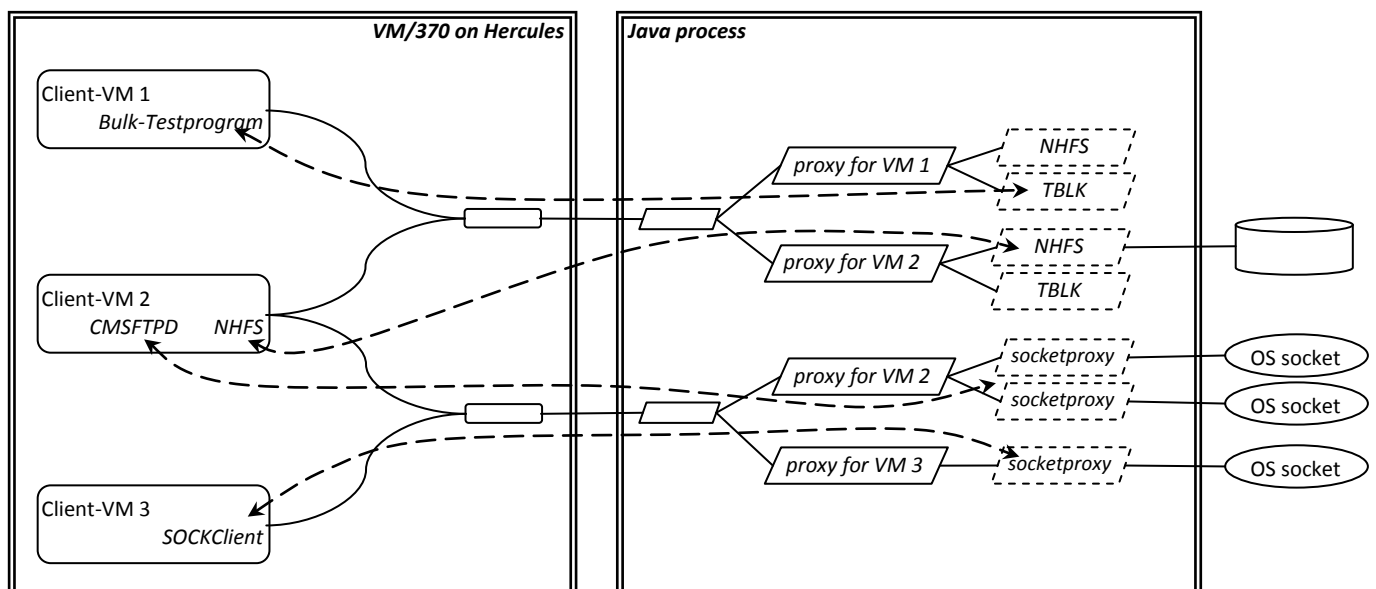


Level-0 services have the advantage of higher flexibility in using the requests/responses, but need a separate gateway for each service. Level-1 services on the other hand only require a single common gateway (NICOFPXY) and can use prebuilt functionality (STDIO-like streams).

The NICOF socket API is implemented as Level-0 service using the gateway TCPIPPXY. The outside proxy manages the sockets used by the CMS client program by providing a socket proxy for each socket on the inside. All operations on an inside socket are delegated by the NICOF socket API to the corresponding socket proxy, which in turn processes these operations on a real socket of the underlying OS where the Java process runs.

The NHFS functionality is implemented as Level-1 service. Other Level-1 services were used while development, for example the TBLK service for testing the NCFIO infrastructure (STDIO-like streams).

The following picture summarizes the communication paths for the standard NICOF setup to support the NHFS and CMSFTPD end user programs:



### 1.3 Performance

Due to the gateway design based on a tn3270 terminal connection, NICOF's communication path will surely not break speed records.

NICOF was developed under Windows on a Laptop with a Core2 Duo @ 2.4 GHz, with Hercules and the external Java process running side by side. Measurements with several use cases gave the following unidirectional transfer rates in this environment:

- Roundtrip test: ~305 Kbyte/s (with ~ 1100 SIO/s and ~ 6.5 MIPS)  
the roundtrip test consists of a CMS program sending a 2048 byte packet and an echo Level-0 service sending back the packet, with the inside program awaiting the response packet before sending the next 2048 byte packet; the test measured the time elapsed for 10240 packet roundtrips.
- FTP binary file GET: ~136 Kbyte/s (with ~ 900 SIO/s and ~ 6.5 MIPS)  
transfer of EREPLIB TXTLIB S using CMSFTPD on the inside and Windows command line FTP client on the outside

- FTP binary file PUT: ~145 Kbyte/s (with ~ 900 SIO/s and ~ 7.5 MIPS)  
transfer of EREPLIB TXTLIB S using CMSFTPD on the inside and Windows command line FTP client on the outside
- NHFS GETBIN: ~171 Kbyte/s (with ~820 SIO/s and ~ 8.2 MIPS)  
transfer of EREPLIB TXTLIB S using NHFS command
- NHFS PUTBIN: ~182 Kbyte/s (with ~870 SIO/s and ~ 8.5 MIPS)  
transfer of EREPLIB TXTLIB S using NHFS command

(these are simple one time measurements with the current NICOF version, not the result of a systematic test series; SIO/s and MIPS are averages over the values indicated by the S/370 screen of the Hercules console)

However, experiments in this and heterogeneous environments revealed 2 problems:

- If both communication partners are not on the same Windows system (separate computers, one or both on Linux etc.), the transfer rates drop by a factor 5..10  
The reason for this was identified as the 3270 TCP/IP connection using default settings for TCP\_NODELAY, which is “off”; in this case, the TCP/IP stack joins small packets to larger ones, sending the resulting packet when it is large enough or no further packets are transmitted in a given time period: this slows down communication if the sender already transmits completely prepared packets (like Hercules), as waiting is superfluous.  
On Windows, the default for TCP\_NODELAY seems to be “on” (or the packet collection mechanism is otherwise bypassed) if both communication partners are on the same machine.
- During mass tests, single 0xFF characters (sent as 0xFF 0xFF) seem (very sporadically) to be interpreted as “record end” 3270 telnet sequence (0xFF 0xEF) in the data transfer outside to inside (i.e. sending a “screen” content), which prematurely terminates a response (bad = data loss) and starts a new transmission, doing some internal handshake with the next byte interpreted as handshake token (worse = communication typically locks up). In both cases, the data transfer will be aborted or will give wrong results.  
This was recognized as a bug in the IAC handling of the 3270 console.

To resolve both problems, 2 fixes to the Hercules source file “console.c” were developed. The fixes are already included in the 4.00 source code tree as of mid-January 2013, so newer Hercules 4.00 binaries will not show these problems.

For the 3.x Hercules versions, a patched version of “console.c” with these 2 fixes is delivered with NICOF 0.6.0, so a corrected version of Hercules can be created (see 2.2.2).

If an unpatched version of a 3.x series Hercules is used to run VM/370 side by side with the external Java process on Windows (or if a slow communication is acceptable on other platforms), the second problem described above can still show up. To be able to use such an original 3.x Hercules on Windows, NICOF supports an encoding scheme avoiding 0xFF characters in the data stream from outside to inside: 7 bytes are sent with the high-bit reset, followed by an 8-th byte having the high-bits of the previous 7 data bytes. Although the data transfer works well when this encoding is used, this slows down the throughput by at least 1/3 or maybe a 1/2 (increasing CPU usage and reducing the SIO/s). This encoding scheme can be activated for a gateway by setting the `useBinaryTransfer` property to `false` (see 6.2.2.1).

## 2 Installing NICOF

There are 3 alternative installation kits, which provide different levels of ease to add NICOF to a VM/370 Sixpack 1.2 system:


































- “Playground” package  
File: `nicof-0.6.0-playground-installation.zip`  
this is the easiest way to explore NICOF and its tools: the playground is an overlay to a freshly unpacked Sixpack 1.2 installation, consisting of an additional disk pack, shadow files for the original VM/370 disk packs, the NICOF Java jar and new configuration and startup files.  
Unpack the Sixpack 1.2 archive, unpack the playground-ZIP, put Java on the PATH and VM/370 with NICOF included is ready to start.
- “Standard” package  
File: `nicof-0.6.0-standard-installation.zip`  
this is a middle-easy installation, consisting of the same disk pack as in playground with a corresponding DIRECT file extract on a tape file, the NICOF Java jar and configuration and startup files.  
The standard setup is intended for extending an existing VM/370 on Hercules installation: unpack the ZIP file in the base directory, extend the Hercules and VM/370 configurations and the startup scripts, put Java on the PATH and NICOF is operational.
- “Manual” package  
File: `nicof-0.6.0-manual-installation.zip`  
this setup allows to extend an existing VM/370 on Hercules installation from the tapes used to build the playground disk pack, where all configuration steps must be performed manually.















The next sections describe the content of the installation packages summarized above and the general requirements to install NICOF. The installation process for each package is the described in detail.

### 2.1 Content of the delivery packages

The following table gives an overview over all files are available with NICOF 0.6.0 and which installation package contains those files.

Playground	Standard	Manual	File / Directory / Description
x			<code>deltas_from_sixpack-1.2.txt</code> <i>A short readme file describing the changes from an original VM/370 Sixpack 1.2</i>
x			<code>hercules.rc</code> <i>The modified .rc file to additionally AUTOLOG the inside proxy VMs NICOFPTY and TCIPPPXY</i>
x			<code>hercules.rc-pre-nicof</code> <i>The renamed original .rc file from Sixpack 1.2</i>
	x		<code>HerculesConfiguration_NICOF-pack-unix.txt</code> <code>HerculesConfiguration_NICOF-pack-win.txt</code> <i>.conf fragment to add the NICOF disk pack to an existing Hercules configuration file (unix resp. windows line ends)</i>

Playground	Standard	Manual	File / Directory / Description
x			 mecaff.jar  mecaff-console.cmd  mecaff-console.sh <i>MECAFF 1.2.5 console and new startup scripts for the console</i>
x	x	x	 nicof-proxies.cmd  nicof-proxies.sh <i>Startup scripts (unix resp. windows) for the external Java process for NICOF</i>
x			 sixpack+mecaff+nicof.cmd  sixpack+mecaff+nicof.conf  sixpack+mecaff+nicof.sh <i>Hercules configuration and startup scripts for VM/370 Sixpack 1.2 including NICOF and optionally the MECAFF console</i>
x	x		 disks
x	x		 vm3350-z.ncfdsk.7f0.cckd <i>Additional 3350 disk pack preloaded with the NICOF users, sources and binaries. As this data does not fill up a 3350 disk, there is enough free space for new users of the VM/370 system</i>
x			 shadows  vm3350-11  vm3350-21  vm3350-31  vm3350-41  vm3350-51  vm3350-61 <i>Shadow files extending a freshly unpacked VM/370 Sixpack 1.2 with DIAG-58, S/381 modification to CP, MECAFF 1.2.5, HX-fix for GCCLIB and users/disks on the NICOF disk pack added to the CP directory</i>
x	x	x	 nhfs  readme.txt <i>Default NHFS base directory where user directories will be created</i>
x	x	x	 nicof  level0_echo.properties  nicof.jar  nicof_logging.properties  nicofpxy.properties  tcpip.properties <i>Java-jar (binary) of the NICOF outside proxy including the default services delivered and configuration files for the outside proxies for the NICOFPPXY and TCPIPPXY gateways</i>
x	x	x	 hercules-3.xx-fix  console.c <i>Patched source for Hercules 3.xx</i>
x	x	x	 src  nicof-0.6.0-cms-sources.aws  nicof-0-6-0-javasrc.zip  nicof-0.6.0-manual.docx <i>The CMS sources for the NICOF APIs and user programs and Java sources for the external Java process, as well the MS Word file for this document</i>
	x	x	 tapes
	x		 nicof-0.6.0-ncfdsk.direct.aws <i>AWS-tape with a DIRECT-file extract defining the disks and VM-users on the NICOF disk pack (CMS TAPE command)</i>

Playground	Standard	Manual	File / Directory / Description
		x	 nicof-0.6.0-cms-binaries.aws  nicof-0.6.0-proxyfiles.aws  nicof-0.6.0-client01.aws  nicof-0.6.0-client02.aws  nicof-0.6.0-client03.aws <i>AWS-tapes (CMS TAPE command) having the NICOFS CMS binaries and the files for the 191 disks of the inside proxies NICOFPXY and TCPIPPXY, as well as the 191 disks for the test users CLIENT01..CLIENT03</i>
x	x	x	 win-testtools  sock_client.c /  sock_client.exe  sock_server.c /  sock_server.exe  tselect_terminate.c /  tselect_terminate.exe  tselect_test.c /  tselect_test.exe <i>Sources and windows binaries for test programs used while developing the TCP/IP functionality, the CMS counterparts are:</i> <ul style="list-style-type: none"> <li>- TSELECT C / TSERVER C (inside) for: tselect_test / tselect_terminate (outside)</li> <li>- SOCKS C (inside) for: sock_client (outside)</li> <li>- SOCKC C (inside) for: sock_server (outside)</li> </ul>

## 2.2 Prerequisites

### 2.2.1 Java runtime

To use NICOFS, a Java runtime version 1.6.0 (Java 6) is required; a newer runtime 1.7 (Java 7) will probably work, but this has not been tested.

The Hercules and Java programs must be on the PATH variable if the delivered scripts are used.

### 2.2.2 Hercules

When using a (reasonably recent) Hercules binary version 4.00, no further actions are required, as the advised patches to `console.c` are already included (since mid-January 2013).

If Hercules in a version 3.x is used, it is advised to apply a patch to the sources to avoid the problems described in section 1.3.

A patched version of the file `console.c` is provided in the subdirectory:

```
nicof/hercules-3.xx-fix
```

The fixes were applied to the file `console.c` from Hercules 3.10, the patches file can be used to replace the original file of same name in any Hercules version since 3.07 (possibly also for earlier versions, but this was not tested).

To rebuild the binaries from sources, at least the ZLIB and LIBBZ2 libraries must be provided for the target platform and configured to be included in the Hercules build.

(see the build instructions <http://www.hercules-390.eu/hercinst.html#instsource> for the target platform)

## 2.3 Installation of the Playground package

Starting point for a playground installation is a freshly unpacked VM/370R6 Sixpack 1.2 base directory, which is extended with a pre-installed standard package (so the steps described in 2.4 have already been done).

As the playground installation will replace the shadow files for the Sixpack 1.2 disk packs, no work should have been done with this system (hence freshly installed), as these work would be lost.

To install the playground, unpack the file

```
nicof-0.6.0-playground-installation.zip
```

in the base directory of the target VM/370 system (some files of the Sixpack 1.2 are replaced, so message boxes about how to handle duplicate files must be confirmed for overwriting).

This will extend the system with a ready-to-run NICOF configuration, consisting of an additional disk pack, a prepared Hercules configuration, startup scripts and shadow files for the standard disk packs. The shadow files already contain the extended CP directory defining the minidisks and users on the additional disk pack (see 2.4) as well as the following components (see also the file `deltas_from_sixpack-1.2.txt`):

- DIAG-58 V1.08 modification for CP
- S/381 V1.0 modification for CP
- GCCLIB with HX fix incl. modified SYSPROF EXEC S2
- MECAFF 1.2.5 static binaries and dynamic binaries for GCCLIB with HX fix and SYSPROFX EXEC Y2
- MECAFF 1.2.5 help files
- MECAFF console 1.2.5

The default NHFS base directory, where the user-directories will be located, is the subdirectory `nhfs` in the base directory of the VM/370 installation. To use a different directory, modify the configuration parameter

```
hostfilesvc.basepath
```

in the file `nicofpxy.properties`.

The playground system is started with the command:

```
sixpack+mecaff+nicof.cmd
```

for as Windows system resp. for UNIXoids:

```
sixpack+mecaff+nicof.sh
```

This will start the NICOF Java process and optionally the MECAFF console (if `mecaff.jar` is present) and the Hercules with the configuration file `sixpack+mecaff+nicof.conf`. The inside proxy VMs are started through the `hercules.rc` file, which executes the necessary AUTOLOG commands.

VM/370 users wishing to use NICOF tools should either add a line

```
EXEC NCFACC
```

to their PROFILE EXEC or enter

```
NCFACC
```

once before using the NICOF tools.

Alternatively, the command `EXEC NCFACC` can be added to the `SYSPROFX EXEC Y2` to provide the NICOF tools to all users.

## 2.4 Installation of the Standard package

To simplify extending an existing VM/370 Sixpack 1.2 installation, the standard package provides the inside software as an additional disk pack, configuration files to integrate the disk pack and the files to run the external Java process for the outside proxies.

The additional 3350 disk pack has the label NCFDSK and holds several minidisks for the following users:

- NICOFSRC  
this pseudo-user provides the NICOFS system disks to other VMs, these disks are:  
191 = R/O disk A for users NICOFPXY and TCPIPPXY (gateway VMs)  
192 = sources disk for NICOFS development  
193 = binaries disk for NICOFS distribution, optionally accessed as disk N by other users
- NICOFPXY and TCPIPPXY  
these are the inside proxy VMs for the standard (Level-1) and the TCP/IP gateways; both users share the NICOFSRC 191 minidisk in R/O mode as disk A with a PROFILE EXEC which automatically starts the gateway if the VM is IPL-ed in disconnected mode.
- CLIENT01, CLIENT02 and CLIENT03  
these are test users used during development, each user having its own 191 disk; these users may also be useful for other purposes.
- NCFDSK  
The user NCFDSK simply reserves the remaining space on the disk to simplify bookkeeping when creating new users on this disk.

Installing the standard package into a working VM/370 Sixpack 1.2 installation starts with unpacking the file

```
nicof-0.6.0-standard-installation.zip
```

in the base directory of this VM/370 system (where the directories `disks`, `io`, `batch` can already be found). This will add the new directories `nicof` and `nhfs` and some helper files to the VM/370 installation and the NICOFS disk pack to the `disks` directory.

The following sections describe the manual steps for integrating the unpacked files into the system.

### 2.4.1 Extending the Hercules configuration for VM/370

#### 2.4.1.1 Disk configuration

Depending on the OS where Hercules runs, add the content of one of the files

```
HerculesConfiguration_NICOFS-pack-unix.txt
```

or

```
HerculesConfiguration_NICOFS-pack-win.txt
```

to the Hercules configuration file for VM/370, typically the file `sixpack.conf` (for a plain Sixpack 1.2 installation) or `sixpack+mecaff.conf` (if MECAFF is also installed). Appending the new configuration lines at the end of the `.conf`-file is sufficient, it is not necessary to place the lines at a specific place in the file.

This works if the real device 7F0 is not already in use. If this is the case, any other device defined as 3350 in DMKRIO can be used instead of 07F0 in the new configuration file (the `.cckd` file needs not to be renamed, the device address in its name is only for documentation purposes).

### 2.4.1.2 3270 devices configuration

Each inside proxy needs a dedicated 3270 device, which will be addressed by the external proxy when dialing to its inside counterpart. For this, a terminal group with single 3270 device must be defined for each gateway, with the group having the name of the inside proxy.

For example, the following line of the original `sixpack.conf` file defining 8 3270 lines at devices 03F8 to 03CF

```
03C8.7 3270
```

can be splitted to reserve 2 lines for the NICOF gateways and keep an unused spare line for a possible additional gateway:

```
03C8.4 3270
03CD    3270    NICOFPXY    # NICOF :: 3270 line dedicated to VM NICOFPXY
03CE    3270    TCPIPPXY    # NICOF :: 3270 line dedicated to VM TCPIPPXY
```

(this is in fact the configuration used in the playground package)

### 2.4.2 Setup in VM/370

When Hercules is started with the extended VM/370 configuration file (see above), the new disk pack will already be available in the system. However, the structure of this disk pack (which CMS minidisks are in which cylinders) must be made known to the system.

For this, the tape file `nicof/tapes/nicof-0.6.0-ncfdsk.direct.aws` has a DIRECT file fragment defining new users and their minidisks on the NICOF disk pack. To setup NICOF, use the following steps:

- Load the tape with the following command in the Hercules console:  

```
devinit 480 nicof/tapes/nicof-0.6.0-ncfdsk.direct.aws
```
- Logon as user MAINT
- Attach and load the tape:  

```
ATTACH 480 TO * 181
TAPE LOAD
```

(this will create the file `NCFDSK DIRECT` on the disk)
- Append the content of the file `NCFDSK DIRECT` to the directory file defining the current users on the system. This can be done with the preferred editor or for example with the following commands (with *yours* being `SIXPACK` on a fresh VM/370 SixPack 1.2 system):  

```
RENAME yours DIRECT A PRE_NCF DIRECT A
COPY PRE_NCF DIRECT A NCFDSK DIRECT A yours DIRECT A
```
- Activate the new directory:  

```
DIRECT yours DIRECT
```

(this will add the users `NICOFSRC`, `NICOFPXY`, `TCPIPPXY`, `CLIENT01`, `CLIENT02`, `CLIENT03` and `NCFDSK` to the CP directory)

### 2.4.3 Starting the NICOF inside proxies

The inside proxy VMs are normally started automatically at system startup time, by implementing one of the following configuration changes:



- Add automatic login commands at the end of the `hercules.rc` file (after `/enable all`), for example:
 

```
pause 1
/autolog nicofpxy nicofpxy
pause 1
/autolog tcpippxy tcpippxy
```
- Extend the configuration of the AUTOLOG1 user to also AUTOLOG the NICOFPXY and TCPIPPXY users. For this, logon as user AUTOLOG1 but do not execute the PROFILE (enter `ACCESS ( NOPROF` on first VM READ state) and add the following lines to the PROFILE EXEC (after the user CMSBATCH is AUTOLOGed):
 

```
CP AUTOLOG NICOFPXY NICOFPXY
CP AUTOLOG TCPIPPXY TCPIPPXY
```

(in both cases, the correct passwords must be used if they were changed in the DIRECT file)

Just for the records, an inside proxy can be also started manually, in the case of NICOFPXY:

- either interactively by logging on as NICOFPXY and running the `RUN$PXY` command (an EXEC)
- or in the background by doing an AUTOLOG for the user NICOFPXY; this can be done in the Hercules console by issuing
 

```
/autolog nicofpxy nicofpxy
```

 or by logging in as user MAINT and issuing the same command from this user:
 

```
AUTOLOG NICOFPXY NICOFPXY
```

#### 2.4.4 Accessing the NICOFP tools

The NICOFP tools (MODULEs for NICOFP programs, C include files and TXTLIB for the APIs etc.) are located on the minidisk 193 of the user NICOFSRC. The content of this minidisk must be made accessible to other users wishing to use NICOFP functionality, either by linking and accessing this minidisk (an EXEC for this is already present on this disk) or by copying the disks content to a public system disk, typically disk Y (19E):

- Login as user MAINT
- Access the Y disk in R/W mode:
 

```
ACCESS 19E Y
```
- Link and access the NICOFSRC disk 193:
 

```
LINK NICOFSRC 193 793 RR
ACCESS 793 N
```
- Then depending on the scenario chosen:
  - either copy the file NCFACC EXEC to disk Y:
 

```
COPY NCFACC EXEC N = = Y2
```
  - or copy all files on disk N (except NCFACC EXEC, which is not needed in this case) to disk Y:
 

```
COPY * * N = = Y2
ERASE NCFACC EXEC Y2
```

Remark: The users on the NICOFP disk (NICOFPXY, TCPIPPXY and CLIENT\*) already have the NICOFP tools disk linked in the CP directory and access it in their PROFILE EXEC.

### 2.4.5 Starting the outside proxies

The Java process with the outside proxies must run in parallel to Hercules running VM/370. The following platform specific scripts to manage the Java proxies process are unpacked to the base directory of the VM/370 installation:

```
nicof-proxies.cmd  
nicof-proxies.sh
```

These scripts take the following parameters:

- `start`  
run the NICOF Java proxies as background process
- `stop`  
stop the background Java process with the NICOF proxies  
Warning: if Hercules is started into a background process, care should be taken that the NICOF Java process is stopped after Hercules ends, so the `nicof-proxies` script should be invoked with the `stop` parameter in the same thread as Hercules.

In the startup script for Hercules, the suitable `nicof-proxies.{cmd|sh}` script should be invoked with the parameter `start` before Hercules is started and with the `stop` parameter after the Hercules command, for example for Windows:

```
call nicof-proxies start  
hercules -f sixpack.conf > Log.txt  
call nicof-proxies stop
```

resp. for UNIXoids:

```
sh nicof-proxies.sh start  
hercules -f sixpack.conf > Log.txt  
sh nicof-proxies.sh stop
```

(note that – compared with the original Sixpack 1.2 scripts – Hercules is not started as background process)

There is no specific startup sequence for the inside and outside proxies, as the outside proxy will try to connect to the inside proxy all 3 seconds until it can successfully DIAL to the inside proxy VM and start up the communication. The outside proxy will also try to reconnect if the connection is lost (the inside proxy crashes or disappears in some other way)

When starting the outside proxy process before the VM/370 is started as above, the following message sequence can appear on the NICOF console window (Windows) or the log file `nicof/nicof.log` (UNIXoids):

```
main INF: CommProxy: [nicofpxy] Connecting...  
main INF: CommProxy: [tcpippxy] Connecting...  
main INF: CommProxy: [nicofpxy] Unable to setup Proxy connection to VM/370-Host 'localhost', Port 3270  
main INF: CommProxy: [tcpippxy] Proxy-VM 'TCPIPPXY' not present or not accepting DIALs  
main INF: CommProxy: [nicofpxy] Proxy-VM 'NICOFPTY' not present or not accepting DIALs  
main INF: CommProxy: [nicofpxy] Connected to Proxy-VM  
main INF: CommProxy: [tcpippxy] Connected to Proxy-VM
```

The outside proxies try to connect in parallel at 0,5 seconds distance; consecutive messages with same content are suppressed, so the above message show the states until a connection is successfully established between the inside and outside proxies.

## 2.4.6 Configuration of the outside proxy

The connection parameters and service configuration for each outside proxy are defined in the properties files `nicofpxy.properties` and `tcpippxy.properties` passed to the outside process.

The default NHFS base directory, where the user-directories will be located, is the subdirectory `nhfs` in the base directory of the VM/370 installation. To use a different directory, modify the configuration parameter

```
hostfilesvc.basepath
```

in the file `nicofpxy.properties`.

(see sections 6.2.2.1 and 6.3.2.1 for available configuration options)

## 2.5 Installation of the Manual package

The main difference between the standard and the manual package is that instead of providing the VM/370 part as additional 3350 disk pack, the tapes used to setup the disk pack are delivered. So additional steps are required to install NICOF inside VM/370, while the steps “exterior” to VM/370 (Hercules configuration, external Java process management) are the same (except for adding the disk pack of course).

### 2.5.1 VM/370 internal configuration steps

The internal configuration involves creating users in the CP directory, load their disk 191 content and loading the NICOF software itself.

#### 2.5.1.1 Required and optional VM users

The following users are required to work with NICOF:

Name	Memory	Minidisks	Purpose
NICOFPTY	3 Megabytes	191 : ≥ 1 Cyl.	Standard inside proxy VM
TCPIPPXY	3 Megabytes	191 : ≥ 1 Cyl.	Inside proxy for socket API

The required content of disk 191 for both users is on the following tape file (CMS TAPE format):

```
nicof/tapes/nicof-0.6.0-proxyfiles.aws
```

The following users are optional to work with NICOF:

Name	Memory	Minidisks	Purpose
CLIENT01	3 Megabytes	191 : ≥ 1 Cyl.	Load test user 01
CLIENT02	3 Megabytes	191 : ≥ 1 Cyl.	Load test user 02
CLIENT03	3 Megabytes	191 : ≥ 1 Cyl.	Load test user 03
NICOFSRC	15 Megabytes	191 : ≥ 16 Cyl. 193 : ≥ 2 Cyl.	User holding the minidisks for the NICOF sources (191 = A) and binaries (193, accessed by other users as N)

The users CLIENT01, CLIENT02 and/or CLIENT03 are only needed if load tests will be performed; the content of their 192 disk is in the corresponding tape file `nicof/tapes/nicof-0.6.0-client0x.aws`.

The user NICOFSRC is required only if the NICOF binaries are to be regenerated from source or if the binaries are not directly copied to disk Y (see 2.5.1.2).

### 2.5.1.2 NICOF binaries installation

The tape file (CMS TAPE format)

```
nicof/tapes/nicof-0.6.0-cms-binaries.aws
```

contains the NICOF binaries and API files.

The easiest way to install the NICOF binaries is to unload the files from the tape onto the public system disk 19E (Y). Care should be taken to specify the file mode Y2 when loading the tape to ensure the files can be read when the disk is R/O, for example as user MAINT:

```
ACCESS 19E Y
ATTACH 480 TO * 181
TAPE LOAD * * Y2
CP DETACH 181
ERASE NCFACC EXEC Y
ACCESS 19E Y/S
```

(the file `NCFACC EXEC` loaded from tape is not needed if the binaries are installed on a public system disk and can be therefore erased)

Alternatively, the NICOF binaries can be installed on a separate disk (for example disk 193 of optional user `NICOFSRC`, see 2.5.1.1). In this case, the file `NCFACC EXEC` should be copied to disk Y (again taking care of the Y2 filemode) and adapted to link the correct NICOF binary disk before accessing it as disk N. All users must then issue the command `NCFACC` before using NICOF tools.

If the binaries are installed on a separate disk instead of a public system disk, this minidisk must be also linked as disk 193 to the users `NICOFPTY` and `TCIPPTY` in the CP directory, as these users will access disk 193 as N by default.

### 2.5.2 External configurations steps

The remaining configuration steps outside of VM/370 are the same as for the standard package, so the actions in the following sections must be performed:

- 2.4.1.2 3270 devices configuration
- 2.4.5 Starting the outside proxies
- 2.4.6 Configuration of the outside proxy

## 2.6 NICOF Source code

The source code for the inside and outside components of NICOF is provided as part of all 3 packages in the directory `nicof/src` in the files:

- `nicof-0.6.0-cms-sources.aws`  
this is a tape file (in CMS TAPE format) with the sources for the inside components (APIs, CMSFTPD, NHFS and IOPROXY modules) and the EXEC files to build the components (`NCFCOMP`) and create the binaries minidisk (`NCFDIST`).  
The FTP server borrows some utilities from the MECAFF package, so the MECAFF source package has to be installed and built in order to rebuild the complete NICOF package.

- `nicof-0-6-0-javasrc.zip`  
this is the Eclipse project directory with the sources and configuration files to build the `nicof.jar` library for the external Java process.
- `nicof-0.6.0-manual.docx`  
this document.

NICOF and the NICOF sources are released to the public domain.

## 3 Using NHFS

NHFS allows to access a subset of the files on the computer running the outside proxy. This service is available when the outside proxy is started with the standard Level-1 configuration (using the `nixofpxy.properties` configuration).

### 3.1 Naming conventions and restrictions for outside files

The NHFS service intentionally imposes some restrictions to the outside files (on the computer where the outside proxy runs) to be visible to the inside.

First: only files under a user-specific base-directory can be accessed. This user-directories are located in the configured base-directory for the NHFS service and are created when an inside VM accesses the NHFS service for the first time.

Second: the name components of the files and directories below a user's base-directory must comply with the CMS naming conventions for files. The name parts may be max. 8 characters long with the same allowed characters as CMS filenames and must be named like `filename.filetype`. This allows to transfer the files between the in- and outside without having to specify 2 file-ids, as the same file-id is used to identify both the inside and the outside files. The name of directories must also be max. 8 characters long and with the same allowed characters as CMS filenames. Files and directories not complying with these rules are ignored by the NHFS service and are simply invisible to the inside.

### 3.2 Configuring the outside service

The base directory for the NHFS service is defined in the outside proxy configuration and must be a directory accessible and writable by the java outside proxy process, see 6.3.2.1 for configuring the NHFS service in `nicofpxy.properties`.

### 3.3 NHFS subcommands

The CMS NHFS command provided with NICOF allows to access files and directories on the outside in the invoking user's base directory.

The format of the NHFS invocation is:

```
NHFS <subcommand> [ id-tokens ] [ ( options ) ]
```

The *id-tokens* parameter is a sequence of name tokens (max. 8 characters in the CMS file naming convention). Depending on the subcommand, the *id-tokens* specify either a file (*fn ft*) or a directory path from the user's base-directory or both (in which case the file-id comes first).

NHFS has the following subcommands:

- `list [ dir1 [ dir2 [...] ] ]`
  - list the files and directories in the specified directory *dir1/dir2/...*
  - if invoked without directory names, the content of the user's base-directory.
- `mkdir dirname [ dir1 [ dir2 ... ] ]`
  - create the new subdirectory *dirname* in *dir1/dir2/...* in the user's base-directory
- `type fn ft [ dir1 [ dir2 ... ] ]`
  - type the content of the outside file *fn.ft* on the console;
  - the file is located in *dir1/dir2/...* in the user's base-directory

- `put fn ft [ dir1 [ dir2 ... ] ] [ ( REPLACE ]`  
`putbin fn ft [ dir1 [ dir2 ... ] ] [ ( REPLACE ]`  
 → copy the inside (CMS) file `fn ft` to the outside file `fn.ft`  
 → the file will be written in `dir1/dir2/...` in the user's base-directory  
 → the subcommand `put` transfers the file in text mode, `putbin` in binary mode  
 → in text mode, the EBCDIC to ASCII translation is done for the bracket EBCDIC charset  
 → if the target file already exists, the `REPLACE` option must be given to overwrite it
- `get fn ft [ dir1 [ dir2 ... ] ] [ ( [REPLACE] [LRECL len] [RECFM x] ]`  
`getbin fn ft [ dir1 [ dir2 ... ] ] [ ( [REPLACE] [LRECL len] [RECFM x] ]`  
 → copy the outside file `fn.ft` to the inside (CMS) file `fn ft`  
 → the file will be looked for in `dir1/dir2/...` in the user's base-directory  
 → the subcommand `get` transfers the file in text mode, `getbin` in binary mode  
 → in text mode, the ASCII to EBCDIC translation is done for the bracket EBCDIC charset  
 → if the target file already exists, the `REPLACE` option must be given to overwrite it  
 → the options `LRECL` and `RECFM` specify the format of the CMS file created  
 → `len` must be 1..255, `x` can be V or F; the defaults are `LRECL 80` and `RECFM V`

## 4 Socket API for CMS and FTP server for CMS

The socket API for CMS allows CMS programs to use sockets provided by the communication infrastructure of the OS where the Java proxy process runs. The NICOF implementation does *not* provide a true TCP/IP stack for VM/370 which is able to use a dedicated network communication device. This means that CMS programs are bound to the capabilities and resources available on the underlying OS:

- CMS programs share the addresses and sockets with the programs running natively on the underlying OS
- As the socket API is provided by the external Java program, a CMS program can only do what the Java program is able to do.

This means for example that the CMS FTP server cannot listen on the standard FTP port if a FTP server already runs on the underlying OS or if the Java process does not run with root-privileges on UNIXoids (as TCP/IP ports below 1024 require root-privileges, with FTP using port 21).

In short: NICOF's socket API is not a TCP/IP stack, it merely allows to participate on the TCP/IP stack of the underlying OS. Not more, but not less.

### 4.1 Supported BSD-like socket API

The header file `SOCKET.H` defines all items necessary to work with the socket API provided by NICOF. The data structures, functions and constants defined there as well as the overall behavior of the implementation try to be close to the general socket functionality provided by the different flavors of sockets APIs (BSD, Linux or other Unices, Windows, ...).

The implementation of the socket API for CMS is provided in the link library `NICOFLIB.TXTLIB` on the binaries disk.

However, the NICOF socket API currently only supports the following subset of a generalized socket API:

- Address families:
  - `AF_INET`
- Socket types:
  - `SOCK_STREAM`
- IP protocol types
  - `IPPROTO_TCP`
- Data structures
  - `sockaddr`
  - `in_addr`
  - `sockaddr_in`
  - `hostent`
  - `fd_set`
  - `timeval`
- Macros:
  - `htons()`
  - `ntohs()`
  - `htonl()`
  - `ntohl()`
  - `FD_ZERO()`
  - `FD_ISSET()`



- `FD_SET()`
- `FD_CLR()`
- API routines:
  - `inet_addr()`
  - `gethostbyname()`
  - `socket()`
  - `connect()`
  - `bind()`
  - `getsockname()`
  - `getpeername()`
  - `listen()`
  - `accept()`
  - `send()`
  - `recv()`
  - `shutdown()`
  - `ioctlsocket()`
  - `select()`
- NICOF specific routines:
  - `closesocket()`
  - `selectX()`
  - `nicofsocket_errmsg()`

Non-blocking operation can be enabled with `ioctlsocket()` for the following socket functions: `connect()`, `accept()`, `send()`, `recv()`.

The NICOF socket API currently has the following known restrictions:

- no integration with the (GCCLIB) C-API  
*this means file descriptors for files and sockets are separate things, so C-API functions cannot be used with sockets (`read()`, `write()`, `close()`) and more specifically: a socket must be finalized with `closesocket()` instead of the C-API `close()` (probably permanent restriction)*
- no DATAGRAM functionality (SOCK\_STREAM and IPPROTO\_TCP only, no `sendto()`/`recvfrom()`)  
*this functionality (SOCK\_DGRAM and IPPROTO\_UDP) is simply not implemented yet, the communication protocol with the external Java proxy is already prepared to some extent; however ICMP-support is very improbable (Java restriction)*
- only IPv4 is supported (address family AF\_INET only)  
*this restriction may be lifted some day*
- no OUT-OF-BAND  
*probably permanent restriction (possible Java restriction)*
- explicit initialization of internal communication  
*as the socket API is based in the NICOF communication path, programs must initialize and deinitialize the NICOF client component by invoking `nicofclt_init()` or `nicofclt_initForSMSGs()` before resp. `nicofclt_deinit()` after using the socket API (see 6.2.1.1)*
- additional error conditions for internal communication  
*as the socket API is based in the NICOF communication path, additional failure possibilities are introduced between the socket endpoints, so NICOF-specific errors can occur (internal proxy not reachable, no external proxy connected or the like), which will be treated like socket API errors (i.e.*

*the invoked routine returns a negative value, with the problem identified in the `errno` resp. `h_errno` global variables)*

The comments in `SOCKET.H` for the supported socket routines extensively describe the error codes in the global variables `errno` resp. `h_errno` that will signal the specific error condition if a routine invocation was not successful as well as further specific restrictions.

## 4.2 CMSFTPD – FTP server for CMS

The CMSFTPD module implements a *simple* and *single-session* FTP server for CMS, supporting the following intended usage scenario:

- inside VM/370, the CMS user starts the CMSFTPD program to allow access to all CMS files visible from this VM
- on the outside, the preferred FTP client program is used to connect to VM/370 to manipulate (upload, download, rename, delete) files on the disks of the user running the CMSFTPD program
- CMSFTPD will automatically terminate when the FTP client disconnects (→*single-session*)
- several CMS users can concurrently run a FTP server for their VM, provided they start CMSFTPD with a different listening port for their “private” FTP server (see command line parameter `-p`)

As some FTP clients use more than one FTP connection for a single user FTP user session, CMSFTPD will accept and service further FTP sessions after accepting the first FTP client connection, but will automatically terminate when the last FTP control connection is closed.

Although more than one FTP connection may be open, CMSFTPD will only process one data operation at a time (→*simple*), so file handling operations (LIST, NLST, STOR, RETR etc.) from different FTP connections are serialized.

To stop CMSFTPD while it waits for the first client connection or for the next FTP client command if a client is already connected, enter the single word

TERMINATE

on the CMS console. The program checks for console user input once a second and discards any input which is not the single (case-insensitive) word TERMINATE.

CMSFTPD accepts most of the RFC-959 FTP protocol commands, but primarily implements the file handling and transfer commands as well as the directory list functionality.

The program simulates a UNIX-like hierarchical file system over all accessed disks of the CMS virtual machine running the program, meaning:

- The directory and file separator is the `/` character
- The directory structure is one level deep, with the disk letters of the accessed minidisks being the directory names below the root `/`
- The usual pseudo-directories `.` and `..` are supported, with `..` always returning to the root, as the directory structure is only one level deep
- The files on a minidisk are identified with a `.` (dot) joining the filename and filetype components of the file's id

- Directory and file names are returned in lower case when listing items (FTP commands LIST, NLST) but are handled as case-insensitive items
- The extended wildcard matching of MECAFF's FSLIST is available when listing items
- If the current directory (FTP commands CWD and PWD) is the root, downloads and uploads of files (FTP commands RETR and STOR) will be directed to directory /a (i.e. disk A), as well as rename and delete operations
- Examples:
  - The CMS file PROFILE EXEC A is identified as: /a/profile.exec
  - The following pattern lists all libraries for GCCLIB on disk S: /s/gcclib.\*lib

When uploading files to the FTP server (i.e. to the CMS file system), modifiers can be specified in addition to the file path, defining the characteristics (RECFM, LRECL) of the file to be created as well as the behavior if the file exists. The modifier is appended to the target file specification in the following format:

{ : | ! } [ { V | F } [ nnn ] ]

with:

- : → create but do not overwrite existing file
- ! → create file, overwriting (replacing) an existing file
- V → give the file RECFM V
- F → give the file RECFM F
- nnn → give the file LRECL nnn (range: 1 .. 255), wrapping at column nnn+1

The default modifier is :V80 (RECFM V, LRECL 80, no overwriting if the target file exists).

Examples for target file-ids with modifiers:

test.memo:v120

- upload the source file to the file TEST MEMO on the current minidisk (or disk A)
- give the file RECFM V and LRECL 120
- do not replace (i.e. abort the upload) the file if it exists

/d/run.pli!f80

- upload the source file to the file RUN PLI D
- give the file RECFM F and LRECL 80
- replace the file if it exists

The CMSFTPD program has the following command line parameters:

-h *hostname*

bind the listening socket for ingoing FTP connections to *hostname*  
Default: 127.0.0.1

-p *port*

listen on port *port* for the ingoing FTP connection  
Default: 21

-pwd *password*

require FTP sessions to login with the name of the VM running CMSFTPD and the specified password to allow file system interactions  
Default: none, i.e. any username and password will be accepted

-ro

treat all minidisks as read-only disks, disallowing any upload, delete or rename

Default: minidisks accessed as writable will allow uploads to these disks through the corresponding directory

-replace

automatically overwrite existing files on file upload

this is useful when using GUI FTP clients supporting Drag&Drop

Default: do not overwrite unless the ! modifier is specified when uploading a file

-override

instead of the transfer mode given by the client (binary or ascii) and the default modifier (V80), use an internal filetype-specific predefined mode / modifier combination;

this can simplify the usage of some GUI FTP clients supporting Drag&Drop, as the transfer mode resp. modifier need not to be specified for each operation;

CMSFTPD uses a hard-coded table with sensible defaults for most filetypes of interest for file transfer

-ignoredashargs

ignore the 1. parameter to FTP cmds LIST, NLST or STAT if it starts with a – (dash) ;

this is useful for FTP clients which automatically send an option like -a or -la before the directory to list (e.g. WinSCP or some Linux FTP clients)

-v

verbose mode, printing FTP commands received and status responses on the VM console<sup>1</sup>

Default: silent mode

The following shows a client side FTP session from a (german) MS-Windows to a CMSFTPD started by CMSUSER:

```
C:\xxx>ftp localhost
Verbindung mit zehost.local.net wurde hergestellt.
220 CMSFTPD ready
Benutzer (zehost.local.net:(none)): cmsuser
331 User name noted, need password.
Kennwort:
230 User logged in, proceed.
ftp> dir
200 PORT command successful
150 Opening data connection
drwxrwxrwx 1 root root 0 2014-01-01 12:00:00 a
drwxrwxrwx 1 root root 0 2014-01-01 12:00:00 b
drwxrwxrwx 1 root root 0 2014-01-01 12:00:00 d
drwxrwxrwx 1 root root 0 2014-01-01 12:00:00 e
drwxrwxrwx 1 root root 0 2014-01-01 12:00:00 f
dr-xr-xr-x 1 root root 0 2014-01-01 12:00:00 s
dr-xr-xr-x 1 root root 0 2014-01-01 12:00:00 u
dr-xr-xr-x 1 root root 0 2014-01-01 12:00:00 y
226 Closing data connection
FTP: 480 Bytes empfangen in 0,05Sekunden 9,06KB/s
ftp> dir /d/*.pli
200 PORT command successful
150 Opening data connection
-rw-rw-rw- 1 root root 240 2011-07-02 21:41:00 hello.pli
```

---

<sup>1</sup> When using the -v parameter, it should be ensured that a MORE... console state does not lock the CMSFTPD operation, for example by closely watching the 3270 console state and CLEARing the screen if necessary or by using the MECAFF console with flow mode enabled or by using a 3215 console for the VM.

```

226 Closing data connection
FTP: 68 Bytes empfangen in 0,06Sekunden 1,15KB/s
ftp> get /d/hello.pli hello.pli
200 PORT command successful
150 Opening data connection
226 Closing data connection
FTP: 88 Bytes empfangen in 0,06Sekunden 1,52KB/s
ftp> put hello.pli /a/test.pli:f80
200 PORT command successful
150 Opening data connection
226 Closing data connection
FTP: 88 Bytes gesendet in 0,02Sekunden 4,40KB/s
ftp> bye
221 Good bye, thank you for using CMSFTPD.

```

C:\xxx>

### 4.3 Socket API implementation

The socket API for CMS is implemented by the C module NCFSOCKET C, the required binary for CMS programs is provided in the link library NICOFLIB TXTLIB on the binaries disk.

The outside implementation is realized as Level-0 service. The main service class is

```
dev.hawala.vm370.commproxy.socketapi.Level0SocketAPIHandler
```

## 5 Doing the load tests

The load test programs used during development are part of the NICOF distribution packages and are almost ready to run when using the playground or standard installation packages (i.e. the additional disk pack with predefined users). The load test consists in repeatedly sending 2048 byte packets from CMS, with the outside “echo” proxy sending back this packet with small changes which are checked by the CMS program.

To work with the load test components, the standard gateway NICOFPTY must be configured to use the Level-0 service class

```
dev.hawala.vm370.commpoxy.LevelZeroDirectEchoHandler
```

instead of the Level-1 services supporting NHFS among others (this means that NHFS will not be usable while running the load tests). A prepared properties file is already part of the delivery.

To change the configuration of the default gateway:

- rename or copy the file `nicofpty.properties` to a meaningful backup name
- copy the file `level0_echo.properties` to `nicofpty.properties`
- if the VM/370 system is running: shutdown VM/370
- restart VM/370 including the external Java proxy process

To start the load test, log on as one of the users CLIENT01, CLIENT02 or CLIENT03 and enter

```
RUN-CLNT count
```

with *count* being the number of 2048 byte packets to send to the outside proxy and to receive back.

The load test can be run simultaneously from all 3 CLIENT-users. The `RUN-CLNT EXECs` of the 3 users wait a different time amount until the test really starts, so it is easy to make the test start simultaneously in 3 terminals.

## 6 NICOF API documentation

The main part of the NICOF components is implemented in C using the native CMS C library GCCLIB of VM/370R6 SixPack 1.2.

To allow accessing VMCF and the 3270 device, a thin assembler layer is required, which allows C programs:

- to handle external and internal interrupts,
- to issue/handle VMCF requests/responses,
- to execute SIO operations,
- to post / wait for ECBs.

This low-level API may be useful for other C programs needing to access these facilities usually interfaced in assembler. The client-APIs for the Level-0 and Level-1 protocols are layered on top of the interrupt API.

The Level-0-API allows the direct asynchronous communication at packet level with an external Java process through an internal proxy.

The Level-1-API takes a more RPC oriented approach, where a function is invoked on a service. The services configured at the external Java process can be queried from the base service providing the basic Level-1 API. The Level-1-API is designed to simplify providing high-level APIs for the services, including stream oriented operations similar to the C file API functions `fread()`, `fwrite()`, `fclose()` etc.

The NHFS client-API `SVC_NHFS` H is an example of such a high-level API, which allows to access files and directories on the host system where the NICOF Java-process runs.

The following table shows the layers of the NICOF APIs and the corresponding header files:

Layer	Header files	Description
Level-1	NCFIO H	Stream operations on stream-objects provided by a high-level service
	NCFBASES H	NICOF base service, providing functionality to query configured Level-1 services and to invoke function on such services synchronously or asynchronously.
Level-0	NICOFCLT H	NICOF low-level API to send a request data packet with metadata to the external process via the internal proxy VM and to receive the response from the external process
Low-level API	INTRAPI H	C-API to use VMCF, external and internal interrupts and issue SIO operations from C programs. This API is used both in the internal proxy and the client API implementations.

The implementation of the APIs is provided in the link library `NICOFLIB` `TXTLIB` on the binaries disk.

### 6.1 Interrupt API for C

The NICOF interrupt API for C is provided by the header `INTRAPI` H, requiring `INTRAPI` TEXT (assembled from `INTRAPI` ASSEMBLY and available in `NICOFLIB` `TXTLIB`) to be loaded with the program.

### 6.1.1 Type shortcuts

Throughout the header file INTRAPI.H, the following shortcuts for C data types are used:

```
typedef unsigned char    _byte;
typedef unsigned short   _half;
typedef unsigned int     _full;
typedef unsigned long long _dblw;
typedef void*            _addr;
```

### 6.1.2 Initialisation

To use the API, the function

```
void intrapi();
```

must be called once in the C program. This also ensures that the component INTRAPI TEXT is automatically looked for when the program is loaded.

### 6.1.3 Handling external interrupts

To handle external interrupts, a handler routine must be defined, having an `int`-array as single parameter. This parameter will be passed with the value in R1 passed to the internal interrupt handler (see CMS macro HNDEXT), e.g. having the PSW at offset 24 (byte-offset 96).

The signature for an external interrupt handler is

```
void (*ExtHandler)(int *intrParams);
```

Handling external interrupts is started by registering the handler routine along with the memory area to be used as C runtime stack for this routine:

```
void enable_ext(ExtHandler handler, int *cstack, int cstacklen);
```

The size of the stack for the interrupt handler must be large enough to hold the deepest invocation chain involved in handling the interrupt<sup>2</sup>.

Handling external interrupts is ended by calling the routine:

```
void disable_ext();
```

### 6.1.4 Handling device (internal) interrupts

A C-routine for handling internal interrupts must have the following signature:

```
_full (*IntHandler)(
    _full deviceAddress,
    _full oldPsw1,
    _full oldPsw2,
    _full csw1,
    _full csw2);
```

When invoked to handle an interrupt:

- `deviceAddress` will hold the device raising the interrupt

---

<sup>2</sup> If both external and internal interrupts are handled, each handler must have a separate stack area, as an internal interrupt may occur while the external interrupt handler is active: with a common stack area, the internal interrupt would destroy the stack of the running external interrupt handler.



- `oldPsw1` and `oldPsw2` will be the high-order and low-order word respectively of the old PSW doubleword
- `csw1` and `csw2` will be the high-order and low-order word respectively of the channel status doubleword for the device

Before handling internal interrupts, the handler routine and the memory area to be used as C runtime stack for this routine must be registered using:

```
void set_devint_handler(
    IntHandler handler,
    int *cstack,
    int cstacklen);
```

The size of the stack for the interrupt handler must be large enough to hold the deepest invocation chain involved in handling the interrupt<sup>2</sup>. The registered routine will handle all device interrupts.

Interrupt handling for a specific device is enabled with:

```
void enable_devint_handling(int dev, _byte *failed)
```

Disabling interrupt handling for a device is done with:

```
void disable_devint_handling(int dev, _byte *failed)
```

The device address `dev` must be in the range 0x000..0x7FF. The flag `failed` will be 0 if enabling/disabling the interrupts for the device was successful or 1 if the operation failed.

### 6.1.5 Posting / waiting for ECBs

An ECB (Event Control Block) is a fullword (`_full`), in general passed to the ECB-routines as reference (pointer) to the ECB.

An ECB is posted (activated) with:

```
void post_ecb(_full *ecb);
```

Waiting for a single ECB is done with

```
void wait_ecb(_full *ecb);
```

Before waiting for an ECB, it should be ensured that it is reset, e.g. by setting the ECB variable to 0.

The routine `wait_anyecb( )` allows to wait for one out of  $n$  ECBs, passing the list of ECB-references:

```
void wait_anyecb(_full **ecblist);
```

The last ECB-reference must have the high-order bit set, while all non last ECB-references must have the high-order bit set to 0. The following macros simplify the creation of an ECB list:

- `ECBLIST_ELEM(ecb)`  
returns the reference to `ecb` as non-final list element
- `ECBLIST_END(ecb)`  
returns the reference to `ecb` as final (last) list element

The parameter for these macros is the ECB variable itself, not the reference to the ECB, for example:

```

_full myecb = 0;
_full myecb2 = 0;
_full *ecblist[] = { ECBLIST_ELEM(myecb), ECBLIST_END(myecb2) };
wait_anyecb(ecblist);

```

### 6.1.6 Timer facility

A single timer facility with a resolution of 10 ms (1/100 second) is available through the routine:

```
void set_timer(_full interval, _full *ecb);
```

After calling the routine, processing continues after the invocation point and the passed ECB will be posted when the specified interval is elapsed. It should be ensured that the ECB variable is reset before `set_timer()` is invoked.

A pending timer can be reset (halted) at any time by invoking:

```
void reset_timer();
```

### 6.1.7 VMCF interface

To allow a virtual machine to use the Virtual Machine Communication Facility (VMCF, see the VM/370 System Programmer's Guide<sup>3</sup>), the NICOF interrupt API defines the necessary data structures and constants as well as the routine to execute the Diagnose X'68' function. The data structures and constants defined in INTRAPI H are named as far as possible like the definitions in the System Programmer's Guide, however using lowercase component names in structures instead of the uppercase (assembler-level) names.

The following data structures are declared:

- **DBLWORD**  
this union type allows to access a VM name as a single doubleword, 2 fullwords or 8 characters; the macro `SET_USER_FOR_CP(DBLWORD, char*)` allows to set a DBLWORD with the 8 first characters of a C string, filling up with blanks if necessary.
- **VMCMHDR, \*VMCMHDR\_PTR**  
this struct defines the data area for the VMCF data received through the external interrupt with the interrupt code 0x4001;  
such a data area – aligned to a doubleword boundary – must be given when invoking the “authorize” VMCF function;  
if SMSGs are to be received, the size of the data area must include the message buffer at the end of the VMCMHDR structure, represented by the message component; the total size of the area must have (at least) 169 bytes (including the VMCMHDR itself).
- **VMCPARM, \*VMCPARM\_PTR**  
this struct defines the parameter to the VMCF diagnose call, which must be aligned on a doubleword boundary

The following constants are defined, based on the descriptions in the VM/370 System Programmer's Guide:

- **VMCMRESP, VMCMRJCT, VMCMPRTY**  
flags for VMCMHDR.V1 (status byte associated with the interrupt message header)
- **VMCPAUTS, VMCPPTY, VMCPMSG**  
flags for VMCPARM.v1 (options associated with a particular subfunction)

<sup>3</sup> For example the document GC20-1807-7 IBM Virtual Machine Facility/370: System Programmer's Guide (Release 6 PLC 17)

- VMCPAUTH, VMCPUAUT, VMCPSEND, VMCPSENR, VMCPSENX, VMCPRECV, VMPCANC, VMCPREPL, VMCPQUIE, VMCPRESM, VMCPIDEN, VMCPRJCT  
values for VMCPARM.vmcfunc (DIAG X'68' subfunction code) and VMCMHDR.vmcfunc (request subfunction code transmitted to the interrupt handler)

The VMCF diagnose is invoked with the following routine:

```
int vmcf_request(VMCPARM_PTR param);
```

Both client and server programs must register an external interrupt handler to receive incoming VMCF requests (server) resp. VMCF responses (client). The interrupt code (halfword at byte offset 98) 0x4001 identifies the VMCF interrupt. The interrupt data is passed through the VMCFHDR area registered with the "authorize" VMCF call.

### 6.1.8 SIO support

To initiate device operations, a CCW data type is defined along with constants and macros to set the subcomponents of a CCW.

The CCW type is defined as unsigned long long to represent the doubleword of the CCW. A CCW resp. a CCW chain (consecutive sequence of CCWs) must start at a doubleword boundary.

The following macros allow to initialize all resp. to set single components of a CCW:

- void **CCW\_Init**(CCW \_ccw, \_byte \_cmd, void\* \_addr, byte \_flags, \_half \_len)  
initialize the CCW \_ccw with the device command \_cmd and the flags \_flags, using the data area at \_addr with the area length \_len
- void **CCW\_SetAddr**(CCW \_ccw, void\*\_addr)  
set the address component of the CCW \_ccw to \_addr
- void **CCW\_SetLen**(CCW \_ccw, \_half \_len)  
set the data area length component of the CCW \_ccw to \_len

The following constants are defined to set the flags of a CCW:

- CCWFlag\_CD (chain data)
- CCWFlag\_CC (chain command)
- CCWFlag\_SILI (suppress incorrect length indication)
- CCWFlag\_SKIP (skip data transfer to main storage)
- CCWFlag\_PCI (program controlled interruption)
- CCWFlag\_IDA (indirect addressing)

The routine

```
_full SIO(_full deviceAddress, CCW *ccwChain);
```

invokes the SIO instruction for the deviceAddress with the specified CCW chain, indicating the outcome of the SIO operation with the following values:

- 0 = operation successful (the condition code was B'1000')
- 1 = CSW was stored (the condition code was B'0100')
- 2 = (sub)channel busy / interrupt pending (the condition code was B'0010')
- 3 = not operational (the condition code was B'0001')

- 4 = other (unknown) state (the condition code was none of the above)

To react on the outcome of a SIO operation, the client program must register an interrupt handler and enable interrupt handling for the device. The status of the device is indicated in the `csw2` parameter of the interrupt handler. The constants `Unit_*` and `Channel_*` define the flags allowing to interpret the unit and channel status.

## 6.2 Level-0 API

### 6.2.1 CMS client-side Level-0 API

NICOF's level 0 client API is defined in the header file `NICOFCLT.H` and represents the basic communication means from the inside to the outside provided by NICOF. Each single communication:

- is initiated by the CMS client program, creating a request instance, filling it with packet data and metadata and sending the request to a proxy-VM
- the client program can directly wait for the response for a request sent, or it can have several requests pending, subsequently waiting for the response to any or a specific subset of pending request
- the proxy-VM transmits the request to the external proxy connected to it
- the external proxy dispatches the request to the handler class, which processes the request asynchronously in a worker thread, allocating the new handler class instance on the first request for a client VM
- when the request is eventually processed, the response is sent to the internal proxy, which forwards it to the client VM

Before using any of the above functionality, the Level-0 API must be initialized to prepare of external interrupt handling and VMCF communication (with the option to handle SMSG messages). Before shutting down the program, the Level-0 API must be deinitialized to disable VMCF external interrupts.

Each NICOF request has a life-cycle with the following (sequential) states:

- *new*  
→ the request has been created and is being prepared to be sent
- *pending*  
→ the request is being or has been transmitted to the inside proxy for processing at the outside  
→ the client program may no longer access the request data
- *available*  
→ the inside proxy has sent the response for the request to the client VM  
→ this response is ready to be received
- *returned*  
→ the client program has received the response and has access to the response data
- *free*  
→ the client has freed the request and may no longer access the request data

Unless specified otherwise, `NICOFCLT` routines returning an `int` will specify the outcome of the operation in the returned value. A value of 0 signals a successful operation, any other value indicates an error, which can be resolved to an error message using:

```
char* nicofclt_errmsg(int code);
```

Higher level API may provide their specific error message resolving routine, which will in general invoke `nicofcflt_errmsg()` for NICOFCFLT errors.

#### 6.2.1.1 *Initializing and deinitializing*

Initializing NICOFCFLT has 2 variants. The plain variant simply initializes the API, registering the external interrupt handler and enabling VMCF operations:

```
void nicofcflt_init();
```

The advanced initializing allows handling incoming special messages sent to the client VM with the SMSG CP command:

```
void nicofcflt_initForSMSGs(SmsgHandler handler);
```

The parameter `handler` must have the following signature:

```
void (*SmsgHandler)(DBLWORD vmcmuse, char *smsg);
```

When receiving a SMSG, `vmcmuse` will be the userid (VM name) sending the message and `smsg` will be the message text (DBLWORD is defined in INTRAPI H, see 6.1.7).

Deinitializing NICOFCFLT disables VMCF operations and unregisters the handler for external interrupts:

```
void nicofcflt_deinit();
```

#### 6.2.1.2 *Preparing a request*

A NICOFC request is represented by the opaque data type

```
request_handle
```

defined as `unsigned int`.

A request in state *new* is created with:

```
request_handle nicofcflt_createRequest(  
    uint userWord1,  
    uint userWord2);
```

The parameters `userWord1` and `userWord2` are the 2 metadata full-words to be sent along with the request data packet.

The data packet can be set with the following routine:

```
int nicofcflt_setRequestData (  
    request_handle h,  
    uint length,  
    const char *data);
```

This routine copies the first `length` bytes of `data` as packet content for request `h`, limiting the packet length to 2048 bytes.

Alternatively, the packet data can be translated at byte level while copying with the following routine:

```
int nicofcflt_setRequestDataXlate(  
    request_handle h,
```

```
uint length,
const char *data,
const unsigned char *xtab);
```

The parameter `xtab` must specify a 256-byte region containing the translation table to use.

### 6.2.1.3 *Transmitting a request and waiting for a response*

A request can be sent to the standard gateway (inside proxy NICOFPXY) with:

```
int nicofcvt_sendRequest(request_handle h);
```

Sending a request to a specific gateway can be done with the following routine, passing the name of the inside proxy to address in parameter `vm`:

```
int nicofcvt_sendRequestTo(request_handle h, char const *vm);
```

After calling either of the above routines, the request is in state *pending* and is processed asynchronously, i.e. processing continues in the invoking program (in the client VM) as soon as the VMCF transmission of the request to the inside proxy is initiated.

To handle the invocation of the outside functionality like a synchronous call, the following variants of the above routines will wait for the response to the request to arrive and receive the response (the request will be in state *returned*):

```
int nicofcvt_sendRequestAndWait(request_handle h);
```

```
int nicofcvt_sendRequestToAndWait(request_handle h, char const *vm);
```

For the more general case of asynchronous processing, one or more requests are sent, working continues in the client VM and eventually the program waits or polls for incoming responses.

The following routine waits for a response to a specific request to become available and will receive the response, i.e. the request will be in state *returned*:

```
int nicofcvt_waitForResponse(request_handle h);
```

Waiting for a response for any request is possible with

```
int nicofcvt_waitForAnyAvailable(request_handle *handlePtr);
```

The value pointer to by the output parameter `handlePtr` will be set with the request to which the response is available; this request is still in the state *available*, i.e. the response must be received with `nicofcvt_waitForResponse()`.

The most general wait routine is `nicofcvt_waitForAnyAvailableX()`, which additionally allows filtering the requests to wait for responses and has an optional timeout for waiting:

```
int nicofcvt_waitForAnyAvailableX(
    request_handle *handlePtr,
    uint filterTag,
    uint timeout);
```

The timeout interval is specified in 1/100 seconds, the timeout is disabled by passing the constant `NO_TIMEOUT`.

The `filterTag` filters the requests taken in account: only responses to requests with the matching filter tag will end waiting. Passing the constant `NO_FILTER` disables filtering the requests waited for. The filter tag of a request can be set and fetched at any time with the following routines:

```
void nicofclt_setFilterTag(request_handle h, uint filterTag);

uint nicofclt_getFilterTag(request_handle h);
```

#### 6.2.1.4 Processing a response

Processing a request requires that it is in state *returned*.

The metadata user words of the response can be accessed with:

```
int nicofclt_getResponseUserWords(
    request_handle h,
    uint *userWord1,
    uint *userWord2);
```

The length of the response packet data can be retrieved with:

```
int nicofclt_getResponseDataLength(request_handle h, uint *dataLen);
```

The following routines allow copying up to `bufferLen` bytes from the response packet data into `buffer` and returning the amount of data copied in `*dataLen`, either from the start of the data packet or the offset given in `from`:

```
int nicofclt_getResponseData(
    request_handle h,
    uint bufferLen,
    char *buffer,
    uint *dataLen);

int nicofclt_getResponseDataFrom(
    request_handle h,
    uint bufferLen,
    char *buffer,
    uint *dataLen,
    uint from);
```

The following variants of above routines allow translation of the packet data while copying (see 6.2.1.2 for parameter `xtab`):

```
int nicofclt_getResponseData(
    request_handle h,
    uint bufferLen,
    char *buffer,
    uint *dataLen,
    const unsigned char *xtab);

int nicofclt_getResponseDataFrom(
    request_handle h,
    uint bufferLen,
    char *buffer,
    uint *dataLen,
```

```
const unsigned char *xtab,
uint from);
```

The following routine allows accessing the single byte at offset `idx` of the response data packet:

```
int nicofclt_getResponseDataByte(
    request_handle h,
    uint idx,
    char *b);
```

Finally, a request must be freed with:

```
int nicofclt_freeRequest(request_handle h);
```

### 6.2.1.5 ASCII – EBCDIC translation

NICOFCLT provides built in tables for ASCII ⇔ EBCDIC translation.

The basic translation tables are provided through the following addresses, which can be used as parameter `xtab` when setting the request packet data resp. retrieving the response packet data:

```
const unsigned char *a2e; /* ASCII => EBCDIC table */
const unsigned char *e2a; /* EBCDIC => ASCII table */
```

The following service routines allow to translate a data block:

```
void nicofclt_ebcdic2ascii(const char *src, int length, char *trg);
void nicofclt_ascii2ebcdic(const char *src, int length, char *trg);
```

## 6.2.2 Java server-side Level-0 API

### 6.2.2.1 Configuration of Level-0 services

The configuration file for the outside proxy of a NICOF gateway is a standard Java properties file passed to the external process on the command line. The following parameters are relevant for a Level-0 proxy:

**host**

hostname of the (real) machine where Hercules (and VM/370) runs  
Default: localhost

**port**

port on hostname to connect to (CNSLPORT in the Hercules configuration file)  
Default: 3270

**vm**

name of the inside proxy VM for this gateway (name of the VM/370 user to dial to)  
Default: NIXOFPXY

**usebinarytransfer**

transfer data in binary from outside to inside?  
true : the Hercules emulator running VM/370 has a bugfixed "console.c" and does not need the 7-of-8 encoding  
false: use the 7-of-8 encoding for an unfixed Hercules emulator (slower)  
Default: true



`level0handler`

full qualified name of the Java class for the Level-0 handler to use

Default: `dev.hawala.vm370.commproxy.LevelZeroToLevelOneDispatcher`

Depending on the `level0handler` Java class, additional parameters can or must be specified, which will be interpreted by this class when the client VM specific instance is initialized.

#### 6.2.2.2 Level-0 API

A Java Level-0 proxy implementation must implement the interface `ILevelZeroHandler`.

Each instance of this interface is initialized by the NICOF infrastructure through the `initialize()` method, which receives:

- the configuration of the outside proxy as `PropertiesExt`-instance, allowing to configure the proxy with further data specified in the properties file for the proxy
- the name of the client VM as `EbcdicHandler`
- the host connection as `IHostConnectorInstance`, to be used to send back the response to the client (see below)
- an `IErrorSink`-instance, which is to be used to communicate exceptions caught in the asynchronous request processing to the main processing thread of the proxy

For each incoming request to a Level-0 proxy, the method `getRequestHandler()` is invoked, which is passed a `IRequestResponse`-instance with the request data sent by the client. The method `getRequestHandler()` must return a `Runnable`, which will be enqueued for asynchronous processing. This `Runnable` must:

- process the request using the data in the `IRequestResponse`-object
- fill the response information into this `IRequestResponse`-object
- and finally enqueue the response for transmission by invoking the `sendResponse()`-method of the `IHostConnector`-object given to the Level-0 proxy, passing the `IRequestResponse`-object.

When communication with the internal proxy is shut down, the `deinitialize()` method is called on each client VM instance of the `ILevelZeroHandler` to free all bound resources.

### 6.3 Level-1 API

#### 6.3.1 CMS Level-1 API

The CMS Level-1 API consists of 2 interfaces:

- The base services API is intended to build custom (high-level) APIs for Level-1 services on the outside, like the NHFS API. A client program will probably not use this API directly, but the high-level APIs encapsulating the base services.
- The NICOF stream IO API provides a STDIO-like interface to data streams created on the outside. A NICOF stream is usually returned by a high-level API built with the base services.

The Level-1 API implementation always uses the inside proxy VM `NICOFPTY` (hard-coded), i.e. all Level-1 custom services share the same gateway.

### 6.3.1.1 *Level-1 base services for building custom (high-level) APIs*

The include file `NCFBASES.H` defines the routines to resolve a name for a configured outside service to the current service-id. This numeric service-id must be passed to the routines for invoking a function on a service, where a service call can be synchronous or asynchronous.

The functions returning an `int` identify the operation outcome with the returned value, with 0 signaling success. The following error codes for base services are defined:

`ERR_INVALID_SERVICE`

no service with the given name is configured for the outside proxy

`ERR_BASESVC_INVCMD`

the function code passed is invalid for the service (was rejected by the service)

`ERR_SVC_INVALIDRESULT`

the invoked service function returned an invalid result (service implementation error)

`ERR_SVC_EXCEPTION`

the invoked service function has thrown an exception (service implementation error)

`NEW_BULK_SOURCE`

the result of the invoked function on the service is a read bulk stream

`ERR_BULK_SOURCE_INVALID`

the invoked service function returned an invalid read stream (service implementation error)

`NEW_BULK_SINK`

the result of the invoked function on the service is a write bulk stream

`ERR_BULK_SINK_INVALID`

the invoked service function returned an invalid write stream (service implementation error)

A service name can be resolved to the current service-id with:

```
int ncfbasesvc_resolve(const char *serviceName, short *serviceId);
```

`serviceName` is the name to resolve, this value is matched case-insensitively with the names given to the services in the configuration of the outside proxy (see 6.3.2.1).

The numeric service-id is returned in `*serviceId` if a matching service is currently configured.

When invoking a service function on the outside, the input resp. out data packets may need to be translated between EBCDIC and ASCII. The invocation routines have a parameter `dataFlags` which controls if the base services API does the data translation for the complete data packets. The translation behavior is specified with the following constants:

`INDATA_TEXT`

the invocation data packet (`inData`) is to be converted from EBCDIC to ASCII

`OUTDATA_TEXT`

the returned result data packet (`outData`) is to be converted from ASCII to EBCDIC

## DATA\_BINARY

both the invocation and the result data packet have to be transmitted unconverted

For the synchronous call (see below), INDATA\_TEXT and OUTDATA\_TEXT can be logically OR-ed to specify that both transmitted packets are to be translated.

An asynchronous call to a service function is started with:

```
int ncfbasesvc_invoke_begin(  
    request_handle *hndl,  
    short    svcId,  
    short    svcCmd,  
    int      inCtlWord,  
    void     *inData,  
    uint     inDataLen,  
    byte     dataFlags);
```

This routine starts invoking the function `srvCmd` on service `svcId`. The invocation parameters are the data packet `inData` (with length `inDataLen`) and the additional metadata value `inCtlWord`.

The request handle from the NICOFLCT API (at Level-0) is returned in `*hndl`. Only the synchronization routines (`nicofclt_waitForAnyAvailable[X]()`, `set/getfiltertag()`) may be applied to request handles returned by `ncfbasesvc_invoke_begin()` to coordinate parallel requests to outside services.

The result of an asynchronous call is awaited and received with:

```
int ncfbasesvc_invoke_end(  
    request_handle h,  
    int     *outCtlWord,  
    void     *outData,  
    uint     *outDataLen,  
    byte     dataFlags);
```

This routine waits for the request `h` to get available and returns the metadata result in `outCtlWord`, filling the `outData` buffer with the output data packet and storing the size of this packet in `*outDataLen`. The `outData` buffer must at least have space for 2048 bytes.

A synchronous call to a service function is executed with:

```
int ncfbasesvc_invoke_sync(  
    short    svcId,  
    short    svcCmd,  
    int      inCtlWord,  
    void     *inData,  
    uint     inDataLen,  
    int      *outCtlWord,  
    void     *outData,  
    uint     *outDataLen,  
    byte     dataFlags);
```

This routine joins `ncfbasesvc_invoke_begin()` and `ncfbasesvc_invoke_end()` into a single call, with parameters with same name having the same meaning.

When the returncode of a service function call signals the creation of a stream by the outside proxy (values `NEW_BULK_SOURCE` or `NEW_BULK_SINK`), the `outCtlWord` returned is a stream-Id identifying the stream at the outside proxy. A `BULKSTREAM` for use with the functions defined in `NCFIO_H` can be created for this stream-Id with:

```
BULKSTREAM ncfbid2s(int streamId, bool isSourceStream, bool isText);
```

The `outCtlWord` returned by the invoked routine is passed as `streamId`. The value of `isSourceStream` specifies if the stream will be read-only (`true`) or write-only (`false`). Passing `isText == true` will create a stream for text content, which will automatically do the EBCDIC/ASCII conversion. Passing `isText == false` will create a stream for binary content.

The routine `ncfbasesvc_errmsg(int rc)` returns the error message for a non-zero return code returned by the Level-1 base services or the `NICOFCLT` API.

### 6.3.1.2 *NICOF stream IO functions*

The functionality for bulk streams returned by custom Level-1 APIs is defined in the include file `NCFIO_H`, providing a `STDIO`-like handling of outside resources. Inspired from their `STDIO`-equivalents, the routine names are like in `STDIO`, but start with an “n” instead of an “f” (e.g. “`nread`” instead of “`fread`”, “`neof`” instead of “`feof`” etc.). The parameter identifying the stream has the opaque type `BULKSTREAM` instead of `FILE*`.

To ease the transfer of text content, the routines intended for string I/O (`ngets()`, `ngetline()`, `nputs()`, `nputline()`) automatically perform the translation between EBCDIC (inside) and ASCII (outside), whereas the routines for block data transfer the content in binary mode, i.e. unmodified.

The following routines are available to client programs:

```
char* ngets(
    char *buffer,
    uint bufferLen,
    BULKSTREAM stream);
```

Read up to `(bufferLen - 1)` text bytes from the stream up to a line end, leaving the line end on the string and terminating the string with a `0x00` char.

The remote bytes are assumed to be ASCII and are translated to EBCDIC.

Returns the buffer pointer if any bytes were read or `NULL` if the stream ended (EOF or the like).

```
char* ngetline(
    char *buffer,
    uint bufferLen,
    BULKSTREAM stream);
```

Read up to `(bufferLen - 1)` text bytes from the stream up to a line end, removing the line end on the string and terminating the string with a `0x00` char.

The remote bytes are assumed to be ASCII and are translated to EBCDIC.

Returns the buffer pointer if any bytes were read or `NULL` if the stream ended (EOF or the like).

```
uint nread(
    void *buffer,
    uint bufferLen,
```

```
bool noWait,
BULKSTREAM stream);
```

Read a binary block of data (without any character set conversion) with up to `bufferLen` bytes. If `noWait == false`, the operation may block until the buffer can be filled up to `bufferLen` bytes; with `noWait == true`, only the data immediately available may be returned (with the semantics of "immediately" being left to the service providing the stream). Returns the number of bytes copied to `buffer`.

```
bool nputs(
char *string,
BULKSTREAM stream);
```

Write a text string to the stream, converting the local EBCDIC bytes to ASCII. Returns `true` if writing was at least started (i.e. `false` if the stream was in a state forbidding a write when the function was called).

```
bool nputline(
char *string,
BULKSTREAM stream);
```

Write a text string to the stream, converting the local EBCDIC bytes to ASCII and appending a newline. Returns `true` if writing was at least started (i.e. `false` if the stream was in a state forbidding a write when the function was called).

```
uint nwrite(
void *buffer,
uint bufLen,
BULKSTREAM stream);
```

Write a binary data block to the stream. Returns `bufLen` if writing was at least started or 0 if the stream was in a state forbidding a write when the function was called.

```
bool neof(BULKSTREAM stream);
```

Check if a source stream has reached the end of stream (no more data to read).

```
void nclose(BULKSTREAM stream);
```

Close the stream.

```
int nerror(BULKSTREAM stream);
```

Return the error code of the last failed operation for the stream. The stream specified state codes are the following `NERR_*` constants (but other codes can also be returned, for example for communication errors):

<code>NERR_NOERROR</code>	
<code>NERR_NOT_SOURCE</code>	(read operation on a write-only stream)
<code>NERR_NOT_SINK</code>	(write operation on a read-only stream)
<code>NERR_EOF</code>	(readable stream ended)
<code>NERR_READERROR</code>	(general read error)

NERR_WRITEERROR	(general write error)
NERR_NOTTEXTSTREAM	(text I/O operation on a binary stream)
NERR_NOTBINSTREAM	(binary I/O operation on a text stream)

```
char* nermsg(BULKSTREAM stream);
```

Return the error message associated with the error of the last failed operation for the stream.

## 6.3.2 Java server-side Level-1 API

### 6.3.2.1 Configuration of Level-1 base and custom services

To define a Level-1 services outside proxy, the parameters `vm` and `level0handler` of the configuration file for the gateway should be left unspecified (defaulted) or explicitly specified to NICOFPX resp. the Java class `dev.hawala.vm370.commpoxy.LevelZeroToLevelOneDispatcher`.

This Java class enumerates the parameters in the configuration file to find the definitions for the Level-1 custom service to instantiate for each client VM. For this, it looks for parameter with the name pattern

**service.n**

where *n* is the service counter, which must start with 0 and must count up continuously; enumeration of services in the configuration file stops at the first gap.

The value of each service configuration must have the following structure:

*service-name : full-qualified-java-classname*

The *service-name* is the string used to lookup the service using the CMS Level-1 API. The *full-qualified-java-classname* will be instantiated and initialized for each client VM and may read further parameters using its own service-name to build the parameter name.

Example: the following property file content defines 2 services `hostfilesvc` and `testbulks`, with `hostfilesvc` have an additional parameter `.basepath` giving the root directory for the NHFS

```
# NICOF Host File Service for the SVC_NHFS client-API (used by the CMS command NHFS)
#
service.0 = hostfilesvc : dev.hawala.vm370.commpoxy.LevelOneFileService
hostfilesvc.basepath = ../nhfs

# (test service) Bulk data transfer test service (see SVC_TBLK H)
#
service.1 = testbulks:dev.hawala.vm370.commpoxy.LevelOneTestBulks
```

(this example is in fact extracted from the `nicofpxy.properties` file delivered with NICOF)

### 6.3.2.2 Level-1 API

The class `dev.hawala.vm370.commpoxy.LevelZeroToLevelOneDispatcher` (which is a Level-0 service) manages the Level-1 services configured for the gateway. A Level-1 service must implement the interface `ILevelOneHandler`.

A Level-1 service instance is initialized by calling the interface method

```
void initialize(
    String      name,
```

```
EbcdicHandler clientVm,
PropertiesExt configuration);
```

which receives:

- the name configured for this Level-1 service, this name can be used to read further configuration data from configuration
- the name of the clientVm as EbcdicHandler
- the configuration of the outside proxy as PropertiesExt-instance, allowing to configure the proxy with further data specified in the properties file for the proxy

The `LevelZeroToLevelOneDispatcher` receives all requests from the inside, interprets the reserved user-word for service and function identification and dispatches each request as `Runnable` as required at Level-0 (see 6.2.2.2), invoking the `processRequest()` interface method of the addressed Level-1 proxy synchronously from the processing thread. The `processRequest()` method has the following signature:

```
ILevelOneResult processRequest(
    short cmd,
    int controlData,
    byte[] requestData,
    int requestDataLength,
    byte[] responseBuffer);
```

which receives:

- the `cmd` identifying the functionality to execute
- the user-word in `controlData`
- the input data packet in `requestData`, where the first `requestDataLength` bytes are valid
- the `responseBuffer`, which may be filled with up to 2048 bytes for the output data packet

The return value `ILevelOneResult` is an abstract interface having several derived subtypes, with the returned sub-interface describing the outcome of the invoked functionality. The following subtypes of `ILevelOneResult` may be returned by `processRequest()`:

- `LevelOneBufferResult`  
this is the "standard" result consisting in a returncode, a controlword and optionally the number of bytes filled into the response data buffer passed to `processRequest()`
- `IBulkSink`  
the invoked functionality has opened a bulk sink data stream, writable from the inside to the outside
- `IBulkSource`  
the invoked functionality has opened a bulk source data stream, writable from the outside to the inside
- `LevelOneProtErrResult`  
the invoked functionality signals an error identified by a returncode

The values contained in the `ILevelOneResult`-subtypes are passed back to the invoking routine on the inside as `outCtlWord`, `outDataLen` and `returncode`.

When the `LevelZeroToLevelOneDispatcher` proxy is shut down, the `deinitialize()` method is called on each instance of the `ILevelOneHandler` to free all bound resources.