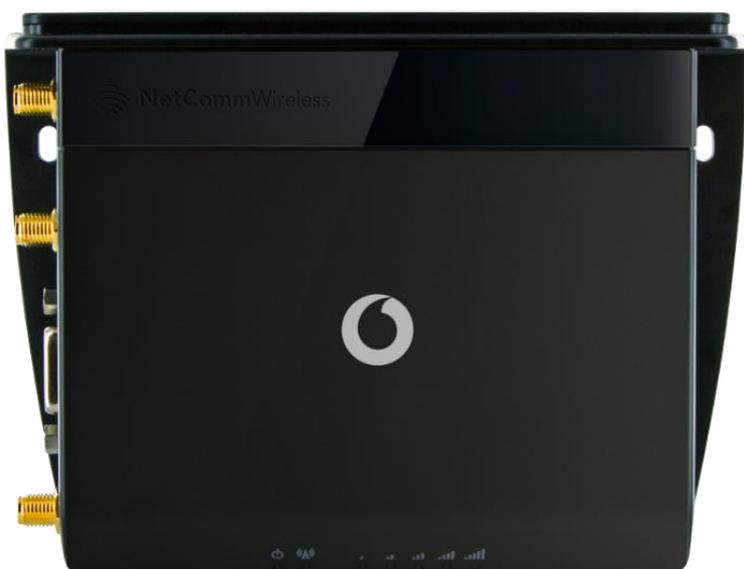


# SDK for Vodafone MachineLink Family



DOCUMENT VERSION	AUTHOR	DATE
1.0 – Initial document release	Iwo Mergler	13/01/2011
2.0 – New layout implementation	Adrian Macarthur-King	16/05/2012
3.0 – Revised Edition	Mace Ledingham	12/06/2012
3.2 – Added Custom Kernel warning	Iwo Mergler	29/11/2012
4.0 – Updated for Vodafone MachineLink 3G	John Manson	22/02/2013
4.1 – Correction to Depends field description on p.8	John Manson	16/07/2013
4.2 – Updated for MachineLink 3G Plus, added new sections on Utilities, Inputs and outputs, Serial port, USB port, SMS and TR-069.	John Manson	22/08/2014
4.3 – Updated librdp description.	John Manson	12/09/2014

# Table of contents

Using the SDK.....	5
Introduction.....	5
Installation.....	5
Dependencies.....	5
Layout.....	5
Initial directories.....	5
Generated directories.....	6
Building.....	6
Examples.....	6
Example Driver.....	6
Example Application.....	8
Adding a new project.....	8
Packages IPKs.....	9
Package Control.....	9
Package structure.....	10
Inserting packages into *.cdi images.....	10
Custom Kernel.....	11
System overview.....	12
Recovery vs. Main.....	12
Boot sequence summary.....	12
Database introduction.....	12
Launching your application.....	12
/etc/inittab.....	12
System start-up scripts.....	13
RDB Templates.....	13
FLASH memory and boot sequence.....	13
Bad block handling.....	14
FLASH partition layout.....	14
ROM-Boot.....	15
Bootstrap.....	15
U-Boot.....	16
Linux Kernel.....	16
File systems.....	17
Watchdog Timer (WDT).....	17
Userspace boot sequence.....	18
System log.....	18
Runtime Database (RDB).....	19
Database Driver (DD).....	19
Library (librdb).....	20
Command line (rdb_get, rdb_set, rdb_wait).....	21
RDB Manager (rdb_manager).....	21
Templates.....	21
LEDs.....	23
Sysfs LED interface.....	23
LED Library (libled).....	24
LED control via command line ('led').....	24
User Interface (Appweb).....	26
System interaction.....	26
Custom Menus.....	27
Utilities.....	28
One Shot Timer Utility.....	28
Usage.....	28
Common Use.....	28
Feedback variable.....	28
Multiple instances.....	29
Absolute time (-a) option.....	29
Arbitrary executable (-x) option.....	29
Implementing a periodic timer.....	29
Other usage considerations.....	30
System control script.....	30
Inputs/Outputs.....	31
Overview.....	31
Hardware Interface.....	31

Wiring Examples.....	32
Open Collector Output driving a relay.....	32
Logic level Output.....	32
LED Output.....	32
Digital inputs.....	33
NAMUR Sensor.....	33
Analogue Sensor with Voltage output.....	34
Analogue Sensor with 4 to 20mA output.....	34
Analogue Sensor with Thermistor.....	34
System Example –Solar powered Router with battery backup.....	35
I/O Design.....	35
Kernel Industrial I/O Subsystem.....	35
I/O Daemon.....	35
Data flow of sensing information.....	35
Configuring the I/O pins.....	36
Using the io_ctl script.....	36
Listing the IOs of the device.....	37
Getting the IO capabilities and setting an IO mode.....	37
Setting pull up status.....	37
Setting pull up voltage.....	37
Writing digital outputs.....	38
Reading digital inputs.....	38
Setting a digital input threshold.....	38
Reading analogue inputs.....	38
Setting hardware gain values.....	38
High frequency input and output.....	38
RDB Variables.....	39
I/O Manager Configuration.....	39
Sensory I/O Configuration.....	39
Extra Information.....	40
Example I/O List (this varies depending on the capabilities of the device).....	40
I/O RDB Structure.....	40
Starting and stopping the IO manager daemon.....	41
Serial Port.....	42
Disabling modem_emulator.....	42
Changing serial port mode (RS232/422/485).....	42
Accessing and configuring the serial port from your application.....	42
Example applications.....	42
USB Port.....	43
Changing USB OTG mode (device/host).....	43
USB Ethernet.....	43
USB Serial.....	43
USB Storage.....	43
SMS.....	44
Sending an SMS.....	44
Receiving an SMS.....	44
Accessing received SMS messages.....	44
Message format.....	44
How to disable SMS messaging.....	45
TR-069.....	46
Using TR-069 with your own set of parameters.....	46
GPS.....	48
Sending GPS coordinates to a TCP server.....	48
List of GPS-related RDB variables.....	49

# Using the SDK

## Introduction

This document serves as a guide to those wishing to develop or customise software to run on a NetComm Wireless product. The SDK itself is a relatively simple wrapper around the NetComm Wireless toolchain. It is intended to simplify the learning curve when compared to the bare cross-toolchain.

As such, it includes the cross-toolchain (binutils, gcc, g++, glibc, etc.), selected sections of the source code, the pre-compiled components of a full system and a set of hierarchical make files.

The SDK includes the kernel source to facilitate the compiling of additional driver modules. Other source code can be supplied on request - most of the system is licensed under BSD, Apache, GPL, LGPL and similar licenses.

## Installation

The SDK can be installed in any directory. Enter the following command to unpack the archive:

```
$ tar -xvjf SDK_Bovine_vdf_nwl10.tar.bz2
```

This will create a directory called SDK\_Bovine\_vdf\_nwl10. The full path to this directory is determined automatically in the top-level Makefile and is available as a variable (BASE) to all project Makefiles.

## Dependencies

The SDK should run on any Linux distribution, but sometimes additional packages may have to be installed. To aid with this, a dependencies.sh script is provided, which can check for missing host system features. Here is an example:

```
$ ./dependencies.sh
Checking make... OK
Checking bash... OK
Checking sed... OK
Checking grep... OK
Checking patch... OK
Checking diff... OK
Checking fakeroot... OK
Checking Bovine compiler... OK
Compiling test program...OK
```



Note: On 64-bit Linux systems, Bovine compiler requires 32-bit libraries installed.

The script only checks for features that were found missing on some systems in the past, but is by no means complete. Should you encounter a missing feature on your host system that is not detected by the dependencies script, please contact NetComm Wireless so that the script can be updated.

## Layout

### Initial directories

DIRECTORY	DESCRIPTION
binaries/	Pre-compiled binaries, including kernel image, main system, rootfs tree.
compiler/	Cross-compiler Toolchain (arm-cdcs-linux-gnueabi).
dependencies.sh	The dependencies script which checks for missing host system features.
libstage/	Contains the include files and libraries required to build applications for the target.
linux/	Kernel source tree.
Makefile	Top-level make file for examples.
projects/	Example projects, you may add your own here.
tools/	Tools used during the build process.

SDK.pdf	This document.
---------	----------------

## Generated directories

DIRECTORY	DESCRIPTION
images/	The results of the build process end up here, together with a copy of the main image.
staging_packages/	Staging directory for packages. All projects install their files here. Each project subdirectory is then later transformed into an *.ipk package.

## Building

To build all projects, just run 'make' in the BASE directory (in this example it is SDK\_Bovine\_vdf\_nw110/). It is not necessary to run this as 'root' and also not recommended.

```
$ make
SDK: Building in dir /home/john/SDK_Bovine_vdf_nw110
mkdir -p /home/john/SDK_Bovine_vdf_nw110/images
cp /home/john/SDK_Bovine_vdf_nw110/linux/config_main_falcon
/home/john/SDK_Bovine_vdf_nw110/linux/.config
cp /home/john/SDK_Bovine_vdf_nw110/binaries/Module.symvers
/home/john/SDK_Bovine_vdf_nw110/linux/
make -C /home/john/SDK_Bovine_vdf_nw110/linux -j8 oldconfig
make[1]: Entering directory `/home/john/SDK_Bovine_vdf_nw110/linux'
  HOSTCC  scripts/basic/fixdep
  HOSTCC  scripts/kconfig/conf.o
  SHIPPED scripts/kconfig/zconf.tab.c
  SHIPPED scripts/kconfig/zconf.lex.c
  SHIPPED scripts/kconfig/zconf.hash.c
  HOSTCC  scripts/kconfig/zconf.tab.o
  HOSTLD  scripts/kconfig/conf
scripts/kconfig/conf --oldconfig Kconfig
...
Packaging /home/john/SDK_Bovine_vdf_nw110/staging_packages/example_app into IPK (target:
/home/john/SDK_Bovine_vdf_nw110/images)
Packaged contents of /home/john/SDK_Bovine_vdf_nw110/staging_packages/example_app into
/home/john/SDK_Bovine_vdf_nw110/images/signal-strength_1.0_arm.ipk
Packaging /home/john/SDK_Bovine_vdf_nw110/staging_packages/example_driver into IPK (target:
/home/john/SDK_Bovine_vdf_nw110/images)
Packaged contents of /home/john/SDK_Bovine_vdf_nw110/staging_packages/example_driver into
/home/john/SDK_Bovine_vdf_nw110/images/example-driver_1.0_arm.ipk
```

When this is done, you'll find a \*.ipk file for every project in images/. As shipped, these would be images/example-driver\_1.0\_arm.ipk and images/signal-strength\_1.0\_arm.ipk.

## Examples

The SDK includes two examples, demonstrating each type of project - kernel drivers and userspace applications.

### Example Driver

This example project demonstrates how to compile a driver module. It is unlikely that this will be necessary in practice, as the kernel contains drivers for all common hardware. In most situations, it is only a matter of enabling the driver in the kernel configuration.

As part of the build preparations, the kernel is partially configured and built, to the extent needed to compile external driver modules. The kernel configuration used here lives in kernel/config\_main\_falcon.

In general, the kernel configuration of the kernel running on the board must match the configuration of the one used to compile the drivers. Should you decide to change this configuration, rebuild the kernel and make sure all the driver modules are recompiled against the kernel tree.

The Makefile in the example\_driver project contains the necessary commands to build the driver and install it into the package staging directory:

```
#
# This is a Makefile to compile external kernel drivers for
# the 2.6-series kernels
#
# Expects the ARCH, CROSS_COMPILE and KDIR variables to be
# set correctly.
#
# The finished driver gets packaged into an ipk

DRIVER := example_driver

obj-m := $(DRIVER).o
```

The obj-m variable sets the name of the driver - NAME.o becomes NAME.ko. In case of a driver with multiple source files (driver1.c and driver2.c), you should set "obj-m := DRIVER.o" and then set "DRIVER-y := driver1.o driver2.o", where DRIVER is the name of your driver.

```
# Current directory
BDIR := $(shell /bin/pwd)

$(info Building against the 2.6 kernel in $(KDIR) )

.PHONY: all
all: $(DRIVER).ko
$(DRIVER).ko:
    make -C $(KDIR) SUBDIRS=$(BDIR) modules
```

This rule invokes the kernel build system to build the driver:

```
.PHONY: clean distclean
clean distclean:
    make -C $(KDIR) SUBDIRS=$(BDIR) clean
    rm -f modules.order
```

For cleaning, too, we invoke the kernel build system:

```
# Generic rule for all other targets
%:
    make -C $(KDIR) SUBDIRS=$(BDIR) $@
```

This rule passes any other targets to the kernel build system. It is not strictly necessary, but may help with debugging.

```
.PHONY: install
install: $(DRIVER).ko
    mkdir -p $(INSTALLDIR)/lib/modules/
    cp $ $(INSTALLDIR)/lib/modules/
    mkdir -p $(INSTALLDIR)/CONTROL
    cp control $(INSTALLDIR)/CONTROL/
```

The install rule gets invoked by the projects Makefile. It has the driver module as a prerequisite, so the module is built first.

The INSTALLDIR variable is set to the absolute path of the package staging subdirectory for this project, so we use it to create an IPK structure (more about [IPKs](#)). In this case, we simply copy the driver into /lib/modules and the IPK control file.

Please note that to make a driver package like this useful, there should also be a script in /etc/init.d/rc3.d that loads the module at boot time.

An example driver is shown below:

```
#include <linux/version.h>
#include <linux/kernel.h>
#include <linux/module.h>

static int __init example_init(void)
{
    printk(KERN_INFO "Example driver: Hello world!\n");
    return 0;
}
```

```
static void __exit example_exit(void)
{
    printk(KERN_INFO "Example driver: Goodbye!\n");
}

module_init(example_init);
module_exit(example_exit);
MODULE_LICENSE("GPL");
MODULE_AUTHOR("<Joe Blogs> engineering@netcommwireless.com");
```

This is an example of a very simple driver. On loading, it emits the "Hello World" string to the system log, on removal it emits the "Goodbye" string. You may test it on the system (after uploading and installing the IPK), by typing "insmod /lib/modules/example\_driver.ko". To remove the driver, type "rmmod example\_driver".

The system log output can be seen using "logread" on the command line or via the Web interface.

## Example Application

The example application (example\_app) is a daemon that displays the 3G signal strength on the LEDs, similar to the "bars" display on a mobile phone. It uses a less accurate, but fast responding signal parameter to make it feasible for antenna adjustment.

This is meant as an example of how to write a daemon and how to interact with the LEDs and the rest of the system. See the [RDB Templates](#) section for a much better way (11 lines of shell script) of achieving the same functionality using RDB templates.

A daemon is an application that, upon launch, forks a child process. The parent process exits while the child process remains running in the background. The required sequence to achieve this is somewhat complex, so we provide a small library (daemon.h / daemon\_lib.a) to make this easier.

When it is running, the daemon watches an RDB (see [Runtime Databases](#) section) variable 'called service.signal\_leds' to determine if its services are required. Only then does it disable the normal LED functions and override them with the current signal strength reading.

Further, the example application includes a web interface extension that allows the user to enable or disable the function. This demonstrates how to add a "user menu" and how to use it to control aspects of the system. Please see the [Custom Menus](#) section for details on this.

This application also demonstrates the use of a [RDB Template](#) to control the daemon via the RDB. The web interface page only sets a RDB variable, which then triggers the template which starts or stops the daemon.

## Adding a new project

The projects Makefile (BASE/projects/Makefile) is written such that it automatically treats all subdirectories of BASE/projects as individual projects. In other words, simply creating a new projects subdirectory with a Makefile is sufficient to hook the project into the build system.

A project Makefile must have, at a minimum, rules for 'install' and 'clean' targets.

A number of variables are pre-set by the build system for the use in project Makefiles. Here is a partial list, the full list can be found at the start of the toplevel Makefile:

```
# ARCH           = [arm] Architecture
# CROSS_COMPILE = [arm-linux-] Cross-toolchain prefix
# PLATFORM      = [Bovine]
# RELEASE       = Base firmware release for this SDK
# BASE          = Base directory of the SDK tree (This_directory/..)
# PBASE        = Projects base (This_directory)
# IDIR         = Image directory, this is where finished packages go
# PSTAGING     = Packages staging directory
# CDCS_INCLUDE = CDCS include file directory
# CDCS_LIBS    = CDCS library directory
# KDIR         = Kernel source tree
```

Additionally, project Makefiles are provided with the INSTALLDIR variable, which is set to the package staging directory of the project (PSTAGING/PROJECTNAME).

The absolute minimum valid package staging directory must contain a CONTROL subdirectory which includes the control file. The control file contains things like package name, package version, target architecture and package description. Optionally, the CONTROL subdirectory may contain install scripts. See the [Package Control](#) section for details.

The remainder of the package staging directory is a representation of the target's root file system. That is, a file installed into "INSTALLDIR/etc/settings" would appear in "/etc/settings" on the target, after the package is installed.

## Packages IPKs

IPK is a package format originally derived from Debian's DPKG. It is still in wide use on embedded systems, although it is now slowly being replaced by OPKG. At the level we are using IPK in the NetComm Wireless development platform, there are no practical differences between the two.

IPK allows dependency tracking (package requires other packages) and thus recursive installs. Most ipkg tool implementations can synchronise package lists with Internet repositories (package feeds) and fetch prerequisite packages from the same repositories. Due to the support of package version numbers, packages may be upgraded automatically.

On the NetComm Wireless platform, packages may be installed via the web interface 'upload' feature.

The system maintains a list of files installed by all packages, such that it can delete them at uninstall time. This can have repercussions if a package replaces existing files. In such a case, the package should rename the original files first, and restore them after the uninstallation.

## Package Control

Beyond the "control" file we have seen in the examples, the CONTROL directory used during the package staging process can contain optional scripts, which control the installation or uninstallation of the package.

The "control file" contains a number of fields. Here is an example:

```
Package: signal-strength
Priority: optional
Section: Misc
Version: 1.0
Architecture: arm
Maintainer: engineering@netcommwireless.com
Depends:
Description: NWL-10 SDK demo application. Displays 3G signal strength via LEDs.
```

FIELD	DESCRIPTION
Package:	Package name, can only contain alphanumeric characters or '-' ([A-Za-z0-9-]). In particular, the underscore character ('_') is used to separate the package name from the version number and is not allowed in the name.
Priority:	This field can be used by the package manager to group packages by importance. Typical entries are "optional", "required", "recommended", etc. Please use "optional" for the NetComm Wireless Development Platform.
Section:	Class of application in package, e.g. Base, Net, Utils, etc. For the NetComm Wireless Development Platform, please use Misc.
Version:	Package version. Typically, two or more numbers separated by '!'. Other formats are admissible, but may confuse the upgrade decision.
Architecture:	For Vodafone MachineLink routers, this is "arm".
Maintainer:	Maintainer of the package, ideally an e-mail address. It is best if this is the person who created the package, rather than the application's author.
Depends:	Comma-separated list of package names this one depends on. If any of these dependencies are missing, ipkg will either refuse installing the package or recursively fetch and install the dependencies first. Leave empty, if package can be installed into a clean SDK target.
Description:	Package description, used by package managers. The description may contain more than one line, but all additional lines must start with a space (' ') character.

Beyond the control file, the CONTROL directory may optionally contain a number of executable programs or scripts - "preinst", "postinst", "prerm" and "postrm".

**preinst** If it exists, this file is executed before the installation. It can be used to rename files that would otherwise be overwritten, or perform additional system checks. If a specific minimum firmware release is required on the target, this is the place to check it. A non-0 return value will abort the installation.

**postinst** If it exists, this file is executed after installation. It is typically used to insert commands into system startup scripts and launch background tasks. The latter is important when creating IPKs that don't require a reboot after installation. It is also possible to request a system reboot here, but if at all possible, this should be avoided.

**prerm** If it exists, this file is executed before uninstalling the package. It can also refuse uninstallation (return non-0), in case the package is not meant to be uninstalled. Typically used to stop background tasks related to the package, in case they would interfere with the uninstallation.

**postrm** If it exists, this file is executed after uninstalling the package. It is typically used to restore original files which were replaced during installation, delete files created by programs in the package, remove commands from startup scripts, etc.

## Package structure

The IPK package is, despite the \*.ipk name, a tar.gz archive. It contains 3 files:

FILE	DESCRIPTION
control.tar.gz	Contains the contents of the CONTROL directory from the staging directory.
data.tar.gz	Contains the remainder of the staging directory (minus CONTROL).
debian-binary	Text file, contains the string "2.0" and is usually ignored.

When a package is installed, the control.tar.gz file is extracted into "/usr/lib/ipkg/info/packagename.\*" and the data.tar.gz file is extracted into the root directory ("/") on the target.

## Inserting packages into \*.cdi images

The NetComm Wireless installable image files (\*.cdi) are, in fact, ISO9660 file system images, containing a number of scripts and image files. In other words, you could rename them to \*.iso and write them to a CD, though it wouldn't make much sense.

It is possible to extract the files from the image via a loopback mount, manipulate the file set and re-package them into a new custom image. In particular, IPK packages may be inserted into the images.

During the main system (see [System Overview](#) section) image install, any IPKs encountered within the image will be installed as well. That is, you can create a custom image which includes and automatically installs a number of IPKs.

IPKs inside the install image are auto-installed in alphabetical order. If the installation order is important, please prepend letters or numbers ([A-Za-z0-9-]) in front of the package name before inserting it.

A script is provided in the BASE/images directory after a build, which allows manipulation of the image contents. In particular, it allows the insertion or removal of any image or \*.ipk file.

For example, here is how you would insert the signal strength demo into the installable image of the NWL-10 (MachineLink 3G):

```
$ sudo ./imagepack.sh vdf_nw110_X.X.X.cdi -a signal-strength_1.0_arm.ipk
Revision: X.X.X
Bovine, main, trunk, Mon Nov 22 17:42:17 EST 2010
-r--r--r-- 2 root root 11796480 2010-11-01 10:52 root.jffs2
-r-xr-xr-x 2 root root 201828 2010-11-01 10:22 u-boot.bin
-r--r--r-- 2 root root 1613347 2010-11-01 10:32 uImage
cdcs_install.sh
cdcs_install_m.sh
cdcs_install_r.sh
checksum.md5
root.jffs2
scripts/
scripts/utills.sh
u-boot.bin
uImage
vars.sh
version.txt
Using CDCS_000.SH;1 for /cdcs_install_r.sh (cdcs_install_m.sh)
Using CDCS_001.SH;1 for /cdcs_install_m.sh (cdcs_install.sh)
73.23% done, estimate finish Tue Nov 23 17:36:17 2010
Total translation table size: 0
Total rockridge attributes bytes: 1466
Total directory bytes: 2048
Path table size(bytes): 26
Max brk space used 21000
6835 extents written (13 MB)
```

The script must be run as root (using sudo), because normal users are not usually authorised to create loopback mounts. You may specify more than one [-a filename] options, to insert several files simultaneously.

The full documentation for imagepack.sh is available at the command line, with "sudo ./imagepack.sh -h".

After the imagepack.sh run, a new image is created (in this case vdf\_nw110\_X.X.X\_C.cdi), which, when installed will also install the IPK.

## Custom Kernel

You may find that your custom application requires changes to the kernel configuration. This may be because you require a kernel feature which is not currently enabled, or maybe an additional driver.

### WARNING:

Changing the kernel configuration may yield a kernel that is incompatible with the driver modules on the running system. Some of these drivers are proprietary, binary only. That is, they can't be rebuilt by the SDK and will fail with an incompatible kernel.

The current kernel has about 1500 options, with more getting added with every release. Enabling everything would result in a kernel that would exceed the available memory on the NetComm Wireless platform.

The default kernel configuration for each product variant is kept in the kernel tree (BASE/kernel), in files like config\_main\_falcon. After performing an initial SDK build, this file will have been copied to .config and the kernel tree will be configured with those settings.

To change them, you need to run "make menuconfig" (requires the ncurses library on the host) or "make xconfig" (requires the QT3 libraries on the host).

However, since this happens outside the scope of the SDK build system, you must set a few environment variables for this. Thus, the complete command line is:

```
ARCH=arm make menuconfig
```

or

```
ARCH=arm make xconfig
```

When you have made the necessary changes, you may compile the kernel and drivers from the top-level directory:

```
make kernel
```

The finished kernel ulmage will end up in arch/arm/boot/ulmage. This ulmage can be inserted into the installable image.

Some configuration options have global scope (cpu, optimisation, debug, etc.). In this case, a config change affects the whole of the kernel, including the driver modules. This is not supported in the SDK, since the binary-only RDB driver won't be automatically rebuilt and may cease to function correctly.

It is highly recommended that, if possible, changes to the kernel should be configured as modules. That way, you only have to extract the relevant \*.ko files from the kernel tree, wrap them into an IPK and install them on the system.

# System overview

The NetComm Wireless platform is designed for ease of development. As such, it has relatively large reserves of FLASH, SDRAM and computing power. It is also quite forgiving of developer errors and can recover from most failures.

## Recovery vs. Main

The NetComm Wireless platform contains two independent systems, each with its own kernel and root file system. These two systems are referred to as "Main" and "Recovery". It is always possible to use one in order to restore the other.

Both systems have Web interfaces that can be used to manipulate the other, inactive system.

Some FLASH partitions are shared between the two systems, most notably the /opt partition which holds uploaded images and the /usr/local partition, which holds the current settings.

## Boot sequence summary

The bootloader (U-Boot) attempts to load the main system by default. If that fails (e.g. missing or corrupt kernel), it loads the recovery system. If that fails too, the bootloader attempts to fetch a recovery system kernel and rootfs from an external TFTP server - on success, those images are automatically flashed.

By pushing and holding the reset button during the start of U-Boot, it is possible to swap the load order of the main and recovery systems. In other words, push and hold the button for 10 seconds and the system will boot into recovery.

Please see the [U-Boot](#) section for details on this.

## Database introduction

Most applications running on a NetComm Wireless system are using the Runtime Database (RDB), either directly or indirectly.

In a nutshell, the RDB is an inter-process communication and storage system, using variables. Processes can create, read or write variables. There is support for user-based access control. Processes can also subscribe to a set of variables and get notified if any of these variables has changed. Most of this functionality is also available to scripts and the command line.

Database variables are identified by name strings, usually in field.field.field notation. Names tend to be self explanatory and can be used to group data into records, e.g. "link.profile.1.auth\_type" and "link.profile.1.status" are variables related to link profile 1.

Database variables can contain arbitrary binary data, but most contain human readable strings. In order to communicate variable values with config files and scripts, URL encoding is sometimes used on non human-readable data. This can waste more storage space than a compact binary representation saves. Thus, it is recommended to use readable strings.

Database variables can also contain flags that describe properties. One important property is "persistent". If set (at variable creation time), the system automatically maintains a copy in the config file. That is, persistent variables survive a reboot.

More detailed information on the RDB can be found in [Runtime Database](#) section.

## Launching your application

Usually, when adding a new application to the system, it needs to be launched somehow, possibly triggered by a system event. The best way usually depends on the application, but here are a few typical options.

To get your new application launched at system boot time, you can either insert a line into /etc/inittab, add a system startup script, or create an RDB template script.

### /etc/inittab

The first task started by the Kernel is Init. Init's job is to launch other tasks as specified in its configuration file, /etc/inittab.

The Init used on the NetComm Wireless platform supports a number of ways to launch applications - by example:

```
::sysinit:/bin/mount
```

This is a system init line, calling the mount command. All lines with the :sysinit: tag will get run in order, each waiting for its predecessor to complete. While there are still :sysinit: lines to run everything else waits.

```
::respawn:/usr/sbin/telnetd  
:once:/etc/init.d/rc.S rc3.d
```

After the :sysinit: lines are done, :respawn: and :once: lines are launched in parallel. The difference is that :respawn: keeps the process running, restarting it if it terminates. :once: will launch the process once and if it terminates, it won't be relaunched.

Another possible tag is :shutdown:, which gets executed, in order, before the system is rebooted or shut down. Please note, however, that it is unwise to rely on a clean shutdown on an embedded system.

Here is a basic example of how to hook your application into inittab. You need to append your new line. However, you must make sure that the procedure can survive a power cut at any time. Thus, this would go into your IPK postinst script:

```
cp /etc/inittab /etc/inittab.new
sync
echo 'null::respawn:/usr/bin/myapp' >>/etc/inittab.new
sync
mv /etc/inittab.new /etc/inittab
```

You may note that we don't create a backup copy. If we did, we could inadvertently kill other packages when we restore it later. To remove the line we added (this goes into the IPK prepm script):

```
grep -v 'null::respawn:/usr/bin/myapp' /etc/inittab >/etc/inittab.new
sync
mv /etc/inittab.new /etc/inittab
```

## System start-up scripts

An application may be launched by a system start-up script. In order to launch your custom application, you may place a script into /etc/init.d/rc3.d which will be executed during boot time.

The system startup scripts are organised within the /etc/init.d/rc.d directory:

DIRECTORY	DESCRIPTION
<b>rc.d</b>	This directory contains the actual system startup scripts. However, they are rarely run here. Instead, the rc2.d and rc3.d directories contain symbolic links to the scripts in rc.d.
<b>rc2.d</b>	The symbolic links in this directory are named such that they can be run in alphabetical order. The scripts linked by rc2.d are run in strict order immediately after boot and are used to set up the critical system infrastructure.
<b>rc3.d</b>	The symbolic links in this directory also have a ordered naming scheme, but must not rely on that ordering. Currently, they are executed in order, but future versions of the system may launch them concurrently.

## RDB Templates

Templates are scripts that are triggered based on system events - changes to RDB variables. Please see the [RDB Templates](#) section for details on templates, this here is only a quick overview.

The templates live in /etc/cdcs/conf/mgr\_templates/. They are mostly shell scripts with a grammar extension that allows them to specify a list of RDB variables they are sensitive to. All templates are run once at boot time and then every time any of the sensitive variables are written - even if the values remain the same.

For example, here is a template that runs every time the first connection profile is connected and the WAN IP address is available:

```
#!/bin/sh

WANIP="?<link.profile.1.iplocal>";
echo '$WANIP' >/tmp/wanip.txt
ftpput some-remote-server.com 3g_ip.txt /tmp/wanip.txt
```

This particular example places any new 3G IP address into a file and uploads that file via FTP to a remote machine - a very primitive way of creating a DynDNS-like system. Of course, to make this reliable the template needs some error handling too.

## FLASH memory and boot sequence

During normal development, it should not be necessary to reflash the board from scratch. This section is provided as a reference only, to enable better diagnosis of any issues that may arise.

## Bad block handling

Due to the nature of NAND FLASH memory, it's unavoidable that the memory may be shipped with a particular small number of "bad blocks", depending on manufacturer and type. The manufacturer marks such bad blocks by writing a specific pattern at a number of alternative locations within the block.

This means that the system must check for such bad blocks and avoid them. Since the manufacturer's test procedure is significantly more sophisticated than what can be done in software on our board, there is no guarantee that we could detect that block as defective or even reliably reproduce the defective mark should the block ever be erased.

It is also possible that normal use of a good block may coincidentally create the same pattern used by the manufacturer's bad block mark - it's only 2 bytes, so 1 in 65536 random numbers will look like a bad block marker.

The third issue arises if a block goes bad during the lifetime of the product. All the file systems used on the NetComm Wireless platform are capable of detecting and handling bad blocks. However, once a block has gone bad, there is no guarantee that we will be able to actually write a bad block marker to it.

To address all those issues, the NetComm Wireless system scans for these bad blocks once, at production time, when the FLASH is empty. The resulting bad block table is written, using two identical copies, to the last two good blocks at end of the device. The bad block table blocks are also marked as bad, so they don't get erased inadvertently. Blocks that go bad later are added to the table in a way that can distinguish between factory marked and system marked bad blocks.

Knowing that a NAND FLASH device may be shipped with a large number of bad blocks, this must be taken into account when partitioning the device. The worst case bad block specification varies between devices, but typical maximum numbers of bad blocks are 20-40, each up to 256K large.

Given the potentially large number of bad blocks in a FLASH device and the fact that those blocks may form a cluster, it would be wasteful to use a fixed partitioning scheme, since every partition (e.g. 1MB bootloader) would require a large number of spare blocks (e.g. 20 = 1.25MB), just in case all bad blocks happen to fall into that partition.

To avoid such wastage, the partitions are created automatically during board production and stored in a U-Boot environment variable. This variable is passed to the kernel at boot time.

Should the U-Boot environment become corrupted or erased, U-Boot is able to re-calculate a suitable partition table. If new bad blocks have developed since production, the resulting partition table will not match the original exactly and the content of some partitions may need to be recreated.

## FLASH partition layout

The NetComm Wireless platform divides the NAND FLASH space into 7 partitions. You may get a list of these partitions on a running system by typing "cat /proc/mtd" on a running system:

```
root:# cat /proc/mtd
dev:   size  erasesize  name
mtd0: 00100000 00020000 "S1S2EN"
mtd1: 00400000 00020000 "rkern"
mtd2: 00b40000 00020000 "rfs"
mtd3: 00400000 00020000 "kernel"
mtd4: 02000000 00020000 "root"
mtd5: 04c00000 00020000 "usr"
mtd6: 07fc0000 00020000 "opt"
```

DIRECTORY	DESCRIPTION
<b>S1S2EN</b>	First and second stage boot loaders (bootstrap & U-Boot) and the U-Boot environment.
<b>rkern</b>	Recovery system Linux kernel.
<b>rfs</b>	Recovery system root file system.
<b>kernel</b>	Main system Linux kernel.
<b>root</b>	Main system root file system.
<b>usr</b>	Shared /usr/local partition, used for configuration files, statistics, etc. Has about 70MB spare storage for customer use.
<b>opt</b>	Shared /opt partition, used for bulk storage. Has about 120MB spare storage for customer use, shared with the upload area (firmware upgrade, etc.) in the web interface.

## ROM-Boot

The processor used for the Vodafone MachineLink routers includes a minimal bootloader contained in on-chip ROM. This bootloader scans the system for memory (NAND, NOR, SRAM, SPI-FLASH, etc) for a valid ARM reset vector, which marks the beginning of all executable code for ARM processors.

This bootloader is very limited and can only access the start of FLASH devices, relying heavily on the device's reset defaults to do so. Since even NAND FLASH devices guarantee that the first sector is non-defective, this always works.

When a valid program is found, the ROM-Boot loads the first 4-8KB of that device into on-chip SRAM and executes it.

If no such program is found (empty flash), ROM-Boot falls back into a very primitive debug mode, using either the serial console port or the USB device controller. In such a case, a full debug cable (serial console + USB device) and associated software must be used to flash the first and second stage bootloaders into FLASH.

## Bootstrap

The first stage bootloader (Bootstrap) lives in the first 8KB of the first eraseblock in the FLASH memory. It is loaded by ROM-Boot into on-chip SRAM and executed.

Bootstrap contains code to start up the SDRAM controller, program the clock generators and knows how to skip bad blocks in FLASH.

After starting the SDRAM controller, it reads the second stage bootloader (U-Boot) into SDRAM and starts it.

## U-Boot

U-Boot is a full-featured boot loader. It knows how to properly handle bad blocks in FLASH, can manage FLASH partitions, understands (some) file systems, includes a scripting language, etc.

U-Boot finalises the system setup and then executes a script that handles the decision making for launching one of the Linux kernels. See the diagram below for a flow diagram of the U-Boot script.

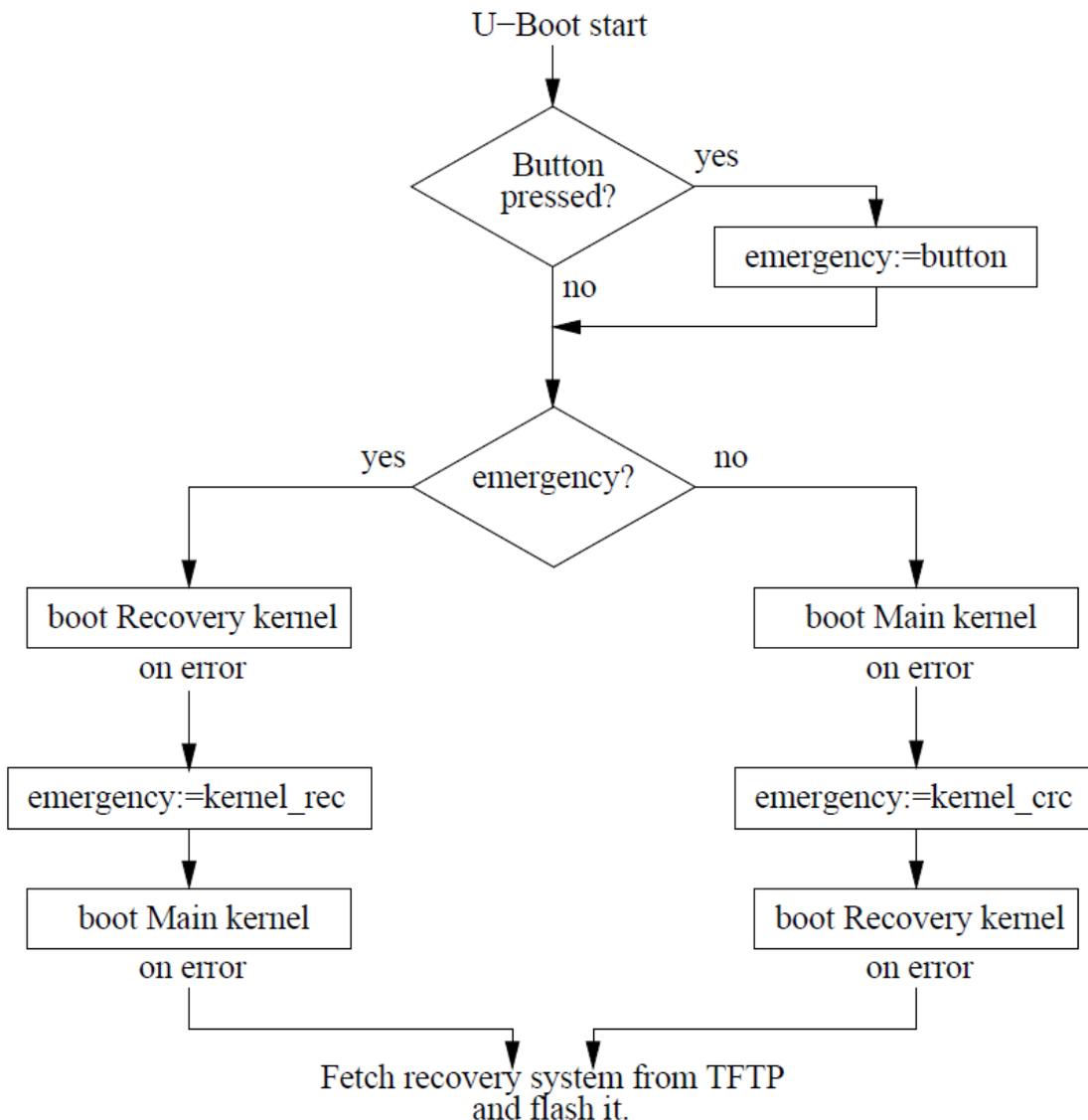


Figure 1: System launch script

Booting a kernel involves determining the kernel command line, which include SDRAM memory size, flash partition layout and root file system to use.

Beyond handling the system startup, U-Boot contains a powerful command line which can be used to reflash parts of the system, manipulate memory locations, talk to the Ethernet PHY, etc.

The U-Boot command line is available on the console serial port, by sending a character during the 3-second boot countdown.

## Linux Kernel

The Linux kernel (either Recovery or Main) is launched by U-Boot. It, in turn, uses the information provided by U-Boot to locate and mount the correct root file system.

On the root file system, the kernel starts the first process, called init. A configuration file (/etc/inittab) specifies which startup scripts to run and what other processes to start.

## File systems

The file systems used on the NetComm Wireless platform are designed to handle sudden power cuts and bad blocks. There are two file systems in use, JFFS2 and UBIFS, depending on the product variant. UBIFS is always used for the two large shared partitions (/opt, /usr/local), while the root file systems are either JFFS2 or UBIFS.

Both file systems are said to use a "log journal" - in other words, any modification to the file system is stored as a patch on the previous content. This allows survival in case of sudden power loss - after the reboot, you either get the old content of the file or the new one, but you won't lose it altogether.

Eventually, the partition is filled up with patches upon patches. Both file systems employ background threads that traverse the partitions, consolidating files, erasing obsolete data and so on. Erasing FLASH memory blocks takes most of the writing time, so it is done in the background, proactively.

This behaviour is invisible most of the time, but there are corner cases where it can surprise the user. For instance, should you run a file writing benchmark, the initial results may be too optimistic. The more free space is available on the partition, the more sustained writes it takes before the background task will have an impact.

Also, both file systems employ data compression. A benchmark program that is using predictable data patterns will give surprisingly good results. If you want realistic worst case behaviour use random or already compressed data.

The job of the background task gets harder as the partition approaches its size limit, since it will get more challenging to find blocks consisting entirely of obsolete data. Frequently, the background task has to collect the non-obsolete bits of a number of erase blocks and consolidate them into a single one to free up blocks for erasure. Both the remaining free space and the pattern of past file operations will affect the performance.

The upshot of all this is that it can be important to select the right partition for the job. We have a selection of file systems and partitions - here are the rules of thumb for use:

PARTITION	DESCRIPTION
<b>rootfs</b>	Separate for Recovery and Main. Use for executables, config files, usually small files that don't change often. JFFS2 has the largest difference between best and worst case behaviour. Writing a single, small file is very fast, but hitting the limits can be painful (minutes to write a small file).
<b>/usr/local</b>	Use for files with frequent, small changes. Log files, statistics files, etc. The system uses this partition to maintain the settings and transfer statistics. Try not to exceed half the partition size to maintain fast response times.
<b>/opt</b>	Use for bulk storage. This partition should be used for large files which are written once and not modified on disk. With this usage pattern, UBIFS tends to show consistent performance over time.

## Watchdog Timer (WDT)

The processor includes a hardware watchdog timer which is able to reset the system, in case of software or hardware error. Software has to regularly reset the watchdog timer to keep it running. If it ever reaches the end of the programmed time interval, it issues a hardware reset.

After reset, the watchdog timer is preset to time out within 16 seconds, the maximum possible interval. The timer can only be programmed once, to either disable it or change the timeout interval. Any further programming access is ignored.

U-Boot issues watchdog resets within its main loop. That is, while U-Boot is busy running the boot sequence, the WDT is reset periodically. When in command line mode, a watchdog reset is only issued when a key is pressed. A command on the U-Boot command line is provided to disable the watchdog (wdtoff), to avoid a reset after 16 seconds of inactivity.

During a normal boot sequence, the kernel watchdog driver becomes active within those 16 seconds and reprograms the watchdog timer for a 2 second timeout. The driver also creates an internal heartbeat timer which resets the watchdog every 500ms. This timer will cease to run if the kernel locks up or crashes, leading to a hardware reset within 2 seconds.

Further, the watchdog driver expects a userspace program to regularly send a watchdog request, to indicate that userspace is still alive. Initially, the timeout for this is set to 90 seconds to allow the boot sequence to complete. After that the timeout is reduced to 15 seconds.

The userspace task (called 'watchdog', part of busybox) opens the /dev/watchdog device and sends a reset command every 2 seconds or so. Should this ever stop, the watchdog driver will let the watchdog timer expire.

## Userspace boot sequence

After the kernel (either recovery or main) has mounted its root file system, the system startup scripts are called. These scripts usually live in the `/etc/init.d/rc.d/` on the unit (or `BASE/binaries/rootfs/etc/init.d/rc.d` in the SDK).

However, the scripts in `/etc/init.d/rc.d` directory are not called directly. Instead, the `/etc/init.d/rc2.d` and `/etc/init.d/rc3.d` directories contain symlinks to the files in `/etc/init.d/rc.d/`. This allows the entries to be ordered - all link names start with `Snnn`, and scripts can be readily inserted into or removed from the boot process. Here is an example of the `/etc/init.d/rc3.d` directory:

```
lrwxrwxrwx 1 iwo iwo 15 2010-11-23 10:18 S300urandom -> ../rc.d/urandom*
lrwxrwxrwx 1 iwo iwo 16 2010-11-23 10:18 S301iptables -> ../rc.d/iptables*
lrwxrwxrwx 1 iwo iwo 11 2010-11-23 10:18 S302usb -> ../rc.d/usb*
lrwxrwxrwx 1 iwo iwo 16 2010-11-23 10:18 S306template -> ../rc.d/template*
lrwxrwxrwx 1 iwo iwo 15 2010-11-23 10:18 S999reflash -> ../rc.d/reflash*
```

At boot time, any links to executable files in `/etc/init.d/rc2.d` are run first, followed by the ones in `/etc/init.d/rc3.d`. The demarcation line between `rc2.d` and `rc3.d` is the local network. At the end of `rc2.d`, the Ethernet interface, telnet login, web UI are all operational.

Additional application packages may add links to their own initialisation scripts into `rc3.d`, as required. In general, scripts should allow a service to be brought up (`scriptname start`) or shut down (`scriptname stop`). However, the NetComm Wireless system will not shut down services automatically, only start them.

## System log

The system log on the NetComm Wireless platform is kept in a ring buffer in memory. This buffer can be read with the 'logread' command. Using the `-t` option to `logread` will attach to the buffer and dump all new entries.

It may be necessary to temporarily switch to logging to a file, to aid debugging. To do this, please replace the line:

```
null::respawn:/sbin/syslogd -n -C256
```

with

```
null::respawn:/sbin/syslogd -n -O /usr/local/logfile
```

This will keep two log files (previous and current) of up to 200KB, rotating them as needed. Please don't use the default `/var/log/messages` as the logfile, since it would deteriorate the root file system performance. If you want, you can set up symlinks in `/var/log/`.

# Runtime Database (RDB)

The Runtime Database (RDB) is the central data repository and inter-process communication system on the NetComm Wireless platform. It allows applications to request or publish information. It also allows applications to synchronise with each other.

Structurally, the RDB is based on a kernel driver, which provides an API via a device node (/dev/cdcs\_DD).

On top of that lives rdb\_lib, a thin library that abstracts the driver API into functions. This library is linked into all applications that communicate with the RDB.

## Database Driver (DD)

Database entries ('variables') are identified by their name. The name is a string containing the characters [0-9a-zA-Z\_0] only. By convention, variables are named in a field.field.etc notation. This allows variables to be grouped logically by membership in specific groups or categories. For instance, here are all SNMP related variables:

```
service.snmp.enable 0
service.snmp.name.readwrite private
service.snmp.name.readonly public
```

Variable content can be an arbitrary block of binary data. However, by convention, data is maintained as human readable strings, mostly single words or numbers. Some database API operations have a performance penalty if arbitrary binary data is used, as such data must be URL encoded / decoded.

Variables are subject to access control permissions, similar to Unix files. There are three permission groups (owner, group & others) and four flags in each group (read, write, erase, perm). By manipulating the permissions, a task can, for instance, create a variable everyone can write to, but only the owner may read (e.g. password check). It is also possible to lock the permissions in place by erasing the perm bit. Since most of the NetComm Wireless system runs as root, this feature is not used much.

Database variables have an additional 'flags' field which denotes special variable properties and behaviour. A few examples:

TRIGGER	DESCRIPTION
<b>PERSIST</b>	Variables with this flag set will be automatically shadowed in the system settings file. That is, they get populated from the settings file at system start and after that, the settings file on disk will mirror the contents.
<b>CRYPT</b>	Variables with this flag set use reversible encryption on their data fields when exported to an external file. This must be used in conjunction with PERSIST, and is set for passwords, PIN numbers, etc.
<b>HASH</b>	Similar to crypt, but uses a one-way cryptographic hash. Rarely used - it can be used to store a password in a non-reversible way. Useful for checking credentials - same passwords result in the same hash.
<b>READ_ONCE</b>	Variable self destructs after one read. Used for temporary keys or other credentials. Avoids inadvertent leaking of critical information.
<b>PERSIST</b>	Variables with this flag set will be automatically shadowed in the system settings file. That is, they get populated from the settings file at system start and after that, the settings file on disk will mirror the contents.

Processes can subscribe to a number of variables. After subscribing, the process may sleep (or do something else) and if any of these variables are written, the process will be notified. This is done either via a signal or via a poll/select call.

The database has a global lock, which can be used to group multiple database operations into a single atomic unit. When a process holds the lock, any other process accessing the database will stop until the lock is released. There is a time limit on the lock (1 second), after which the offending process will be killed.

## Library (librdb)

The RDB library abstracts the driver interface for applications. The relevant include file is `rdb_ops.h`, included in the SDK. The library does not cover the full driver functionality yet and is subject to future extension.

The following functions are supported:

```
int rdb_open(const char *dev, struct rdb_session **s);
```

Opens the database at the device node `dev` and returns a session reference `s` that is used by other RDB functions. Returns a negative value on error.

```
void rdb_close(struct rdb_session **s);
```

Closes the database session. All pending subscriptions are dropped. This is also the only way to delete variable subscriptions.

```
int rdb_getnames(struct rdb_session *s, const char *szName, char *pValue, int *pLen, int nFlags);
```

Search function, returns a list of variable names that include the `szName` substring.

```
int rdb_fd(struct rdb_session *s);
```

Returns the current file descriptor for direct driver access. Returns a positive value if the database is open, negative otherwise.

```
int rdb_subscribe(struct rdb_session *s, const char *szName);
```

Adds a variable to the task's subscription list.

```
int rdb_getinfo(struct rdb_session *s, const char *szName, int *pLen, int *pFlags, int *pPerm);
```

Returns additional variable information - data length, flags, permissions, owner PID and GID.

```
int rdb_set(struct rdb_session *s, const char *szName, const char *szValue, int len);
```

Write value `szValue` of length `len` to database variable `szName`. The variable must exist.

```
int rdb_set_string(struct rdb_session *s, const char *szName, int *szValue);
```

Writes a value `szValue` to the RDB variable `szName`, which should already exist. The string should be 0-terminated.

```
int rdb_get(struct rdb_session *s, const char *szName, char *pValue, int nLen);
```

Read content of a variable. `nLen` is size of `pValue` buffer, returns real variable length.

```
int rdb_get_int(struct rdb_session *s, const char *szName, int *pValue);
```

Reads a variable value and converts to integer.

```
int rdb_setflags(struct rdb_session *s, const char *szName, int nFlags);
```

Writes the flags for a single variable in the database.

```
int rdb_getflags(struct rdb_session *s, const char *szName, int *nFlags);
```

Reads the flags for a single variable in the database.

```
int rdb_create(struct rdb_session *s, const char *szName, const char *szValue, int nLen, int nFlags, int nPerm);
```

Create a new database variable `szName` and write an initial value `szValue` to it.

```
int rdb_create_string(struct rdb_session *s, const char *szName, const char *szValue, int nLen, int nFlags, int nPerm);
```

Create a new database variable `szName` and write an initial value `szValue` to it. The value should be a 0-terminated string.

```
int rdb_delete(struct rdb_session *s, const char* szName);
```

Remove variable from database. This is different from a 0-length value.

```
int rdb_update(struct rdb_session *s, const char* szName, const char* szValue, int nLen, int nFlags, int nPerm);
```

Creates variable if it doesn't exist, writes value. Flags, permissions, etc. are only used during variable creation.

```
int rdb_update_string(struct rdb_session *s, const char* szName, const char* szValue, int nLen, int nFlags, int nPerm);
```

Creates variable if it doesn't exist, writes value. Flags, permissions, etc. are only used during variable creation. `szValue` should be a 0-terminated string.

```
int rdb_lock(struct rdb_session *s, int nFlag);
```

Takes the database lock. Must be released within one second.

```
int rdb_unlock(struct rdb_session *s);
```

Releases the database lock.

## Command line (rdb\_get, rdb\_set, rdb\_wait)

A command line tool is provided to manipulate the RDB, for use within scripts and on the system's command line. This is a single binary file that may be called using one of the three names: rdb\_get, rdb\_set and rdb\_wait.

You may use rdb\_get to fetch a list of variables, or read a single variable. Here we get the list of all variables that contain the string "snmp":

```
root:# rdb_get -L snmp
service.snmp.enable 0
service.snmp.name.readwrite private
service.snmp.name.readonly public
```

You can get a full list of all database variables using "rdb\_get -L | sort".

Creating a new variable (use -p to create a persistent variable - it will end up in the config file):

```
root:# rdb_set testvar test
```

Reading it back:

```
root:# rdb_get testvar
test
```

You can also wait for a variable to be updated:

```
root:# rdb_wait testvar && echo "Got `rdb_get testvar`" &
```

This launches a background task waiting for the variable. Now set it:

```
root:# rdb_set testvar 'new value'
root:# Got 'new value'
[2] + Done          rdb_wait testvar && echo "Got `rdb_get testvar`"
```

## RDB Manager (rdb\_manager)

The system employs a daemon that manages system-wide database related operations. It operates the template system (see the [RDB Template](#) section), manages the system settings file and maintains the backing store for statistics.

Those three functions are launched separately during boot time, so you will see "/usr/bin/rdb\_manager -c", "/usr/bin/rdb\_manager -s" and "/usr/bin/rdb\_manager -t" in the process table.

## Templates

Templates are a way of triggering an action based on a change in system state, as reflected by the RDB.

In a nutshell, a template is a script. This can be any scripting language or configuration file. A simple grammar extension denotes tags containing RDB variable names which are used to trigger the script.

The grammar to mark RDB variables as triggers for the template is "?<rdb.variable.name>:". Several such entries may exist in the template - in which case it becomes sensitive to all. Before the template script is run, these variable placeholders are replaced by the variable values. This way, "?<link.profile.1.iplocal>:" becomes "10.247.49.28".

A template, like any other script, can also use RDB variables without being sensitive to them. The invocation in that case would be "`rdb\_get link.profile.1.iplocal`". This is useful if a template controls a service. In such a case, the template is only sensitive to one variable (e.g. 'service.enabled'), but then uses further variables for configuring the service.

At system start time, rdb\_manager parses all available templates in /etc/cdcs/conf/mgr\_templates and makes note of the database variable mentioned in each one. After that, each template is run once.

Every time one of these variables changes, rdb\_manager creates a copy of the template, replacing those variable tags with the variable content. Then the resulting file is executed.

Templates can set RDB variables and therefore trigger themselves or other templates. It is possible to create loops this way, which will cause high CPU loads. Please be mindful of this possibility.

A template script must terminate quickly. If it needs to start a lengthy process, it may launch it into the background. The template script must then contain code to manage this background process, starting and killing it as necessary.

Here, as an example the simplest template in the system (/etc/cdcs/conf/mgr\_templates/reboot.template):

```
#!/bin/sh

REBOOT_ENABLE=?<service.system.reset>;

if [ "$REBOOT_ENABLE" == "1" ]; then
    rdb_set system.reset 0
    (sleep 10; /sbin/reboot;)&
fi

exit 0
```

The string "?<service.system.reset>" makes this template sensitive to a variable called "service.system.reset". This variable is used throughout the system to request a controlled system reboot. You can trigger a system reboot from the command line, using "rdb\_set service.system.reset 1".

When the variable is set, that line in the template becomes "REBOOT\_ENABLE=1" in the resulting script. Rdb\_manager then runs the script. The script checks the variable value, such that only a value of 1 actually triggers a reboot (you can issue "rdb\_set service.system.reset 0" all day long).

Finally it launches the reboot command as a background process. The reason for this is to allow the system activity to slow down before the reboot. Thus, a delay is used, in this case 10 seconds. The template is not allowed to go to sleep for that long, so the sleep, together with the reboot command is launched as a background task.

It should be clear by now that the example application (display 3G signal strength on LEDs) in the SDK could be much more simply implemented by replacing the led.template with the following:

```
#!/bin/sh

SIGNAL=?<wwan.0.system_network_status.RSCPs0>;

LEDS="`led info LIST`"
dBm=100
for LED in $LEDS; do
    if [ $SIGNAL -lt $dBm ]; then
        led set $LED trigger none brightness 255
    else
        led set $LED trigger none brightness 0
    fi
    dBm=$((dBm-7))
done
```

# LEDs

The LEDs are available via the standard sysfs interface. When using them for custom applications, care must be taken to avoid conflicts with existing functionality.

In general, normal operation of the LEDs tends to operate via triggers (see below), which can be switched off. There are a few cases where LED state is also controlled or modified within RDB templates, e.g. RSSI and DCD LEDs.

For most simple applications, it is sufficient to update the LED state about once per second, thus overriding the templates. Beyond that, it may be necessary to disable the relevant templates.

## Sysfs LED interface

The Vodafone MachineLink routers have 7 LEDs, which are all accessible via the sysfs interface. Each LED is represented by a directory in `/sys/class/leds/`. Starting at the red LED, the names are 'power', 'wwan', 'dcd', 'service' and 'rssi', respectively.

Each LED subdirectory contains a few mandatory files and a number of optional files which appear and disappear depending on the selected trigger.

In a nutshell, a trigger is a kernel level signal that can be used to control an LED. These triggers tend to be attached to specific functions, like hard drives. Triggers can be assigned to LEDs, usually any LED can be attached to any trigger.

Available triggers can be listed as follows - they are the same for all LEDs. Not all triggers are used.

```
root:/sys/class/leds/power# cat trigger
[none] nand-disk timer heartbeat cdcs-wwan cdcs-wlanap cdcs-wlanst
cdcs-pots default-on
```

Without a trigger (trigger=none), the led can be switched on or off by writing 255 or 0 to 'brightness'.

```
root:/sys/class/leds/power# echo 255 >brightness
root:/sys/class/leds/power# echo 0 >brightness
```

A trigger can be set by writing the name of the trigger to the 'trigger' file.

```
root:/sys/class/leds/power# echo heartbeat >trigger
root:/sys/class/leds/power# cat trigger
none nand-disk timer [heartbeat] cdcs-wwan cdcs-wlanap cdcs-wlanst
cdcs-pots default-on
```

TRIGGER	DESCRIPTION
<b>none</b>	No trigger selected, LED will only answer to brightness level changes
<b>nand-disk</b>	Triggers on NAND FLASH activity. Currently not enabled.
<b>timer</b>	Pseudo trigger, makes the LED flash. It creates 'delay_on' and 'delay_off' files which control the flashing on and off times (in ms).
<b>heartbeat</b>	Trigger linked to the system load. It creates a double flash (like a heartbeat), the period of which is related to the system's load average. When the system works harder, the heartbeat increases.
<b>cdcs-wwan</b>	This is the WWAN / 3G trigger. It flashes the LED during 3G activity. Like all cdcs-* triggers, the LED operation can be inverted (by setting brightness_on & brightness_off), which allows link/activity functionality.
<b>cdcs-wlanap</b>	WLAN link/activity. Only used on devices with WLAN access point.
<b>cdcs-wlanst</b>	WLAN link/activity. Only used on devices with WLAN access point.
<b>cdcs-pots</b>	POTS/SLIC interface activity. Only used on devices with phone interface.
<b>default-on</b>	Pseudo-trigger, when set switches the LED on. The kernel contains default trigger settings for all LEDs, to be used during boot time. This trigger allows a LED to be solid on during boot.

## LED Library (libled)

This library is provided to simplify the LED operations. It accesses the LEDs via the sysfs files as well, but packages these accesses into function calls.

The library allows for LED 'properties' to be read or set. A property is a name+value pair which matches the name of a sysfs file and the current or future value in that file. See libled.h for usage details.

## LED control via command line ('led')

The first thing to do before manipulating the LEDs manually is to set the LED Display Daemon to User Mode:

```
root:~# ctrl_dispd.sh user_mode
```

This enables you to take full control of the LEDs without the display daemon interfering with the LED status. To control of the LEDs to the router once more use the following command:

```
root:~# ctrl_dispd.sh router_mode
```

The 'led' command line program is a simple command line wrapper around libled. It allows writing of cross-platform, portable shell scripts. You can use it to determine the number of available LEDs and the physical order:

```
root:# led info NUM
5
root:# led info LIST
power wwan dcd service rssi
```

The program may also be used to save, restore and manipulate LED state. For example, the following command line gets the power led to flash and then restore the old state:

```
root:# STATE=`led save power`
root:# echo $STATE
power trigger none brightness 255
root:# led set power trigger timer delay_on 100 delay_off 100
```

Now, we restore the old state:

```
root:# led set $STATE
```

Each LED may display red, green or amber with amber being achieved by setting the LED to display both red and green at the same time. For example, you can set the power LED on the device to red by using this command:

```
root:~# led set power_r trigger none brightness 255
```

The LED can then be set to amber by also setting the power LED to display green:

```
root:~# led set power_g trigger none brightness 255
```

The following script is installed on the system (leds.sh) and is a good example of portable LED control. It creates a LED scanning effect, using the timer trigger as a PWM to 'fade' the LEDs. When terminated, the old LED states and functions are restored.

```
root:# cat /bin/leds.sh
#!/bin/sh
# LED demo
#

led save ALL >/tmp/ledsave

N=`led info NUM`
LIST=`led info LIST`

echo "Leds: $N, $LIST"

pick() { shift $1; echo $1; }

# $1 number, @$ val
ledset() {
    L=$1; shift
    if [ $L -lt 0 ] || [ $L -ge $N ]; then return; fi
    LED=`pick $((L+1)) $LIST`
    led set $LED @$
}
}
```

```
exiting() {
    echo "Caught signal, exit"
    if [ -e /tmp/ledsave ]; then
        led restore /tmp/ledsave
        rm -f /tmp/ledsave
    fi
    exit 0
}
trap 'exiting' EXIT
trap 'exiting' INT
trap 'exiting' TERM

X=-2
D=1
while true; do
    ledset  $$(X-2)$  trigger none brightness 0
    ledset  $$(X-1)$  trigger timer delay_off 10 delay_on 1
    ledset $X trigger none brightness 255
    ledset  $$(X+1)$  trigger timer delay_off 10 delay_on 1
    ledset  $$(X+2)$  trigger none brightness 0
    X= $$(X+D)$ 
    if [ $X -ge  $$(N+2)$  ]; then
        D=-1;
    fi
    if [ $X -le -2 ]; then
        D=1;
    fi
done
```

# User Interface (Appweb)

The Web-based user interface is currently built around the Appweb server. This server supports server side scripting and is linked with the RDB system, such that server side (ESP) scripts have access to RDB variables.

Web interface related files are located in the /www directory, which acts as the web server root.

## System interaction

Most pages use ESP to communicate with the system via RDB variables. Those RDB variables then control various aspects of the system via the RDB templates. Complex settings are validated via Java Script inside the respective pages.

The Web UI page of the example application demonstrates how to use ESP to access an RDB variable:

```
<%if( request['SESSION_ID']!=session["sessionid"] ) redirect('/index.html');%>
```

This makes sure the user is redirected to the log-in page if the user is not logged in yet.

```
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
<title>Signal Strength LEDs</title>
<link href="/BovineStyle.css" rel="stylesheet" type="text/css" />
```

The look-and-feel of the page is defined here. Using the style sheet takes care of the page layout.

```
</head>
<!-- Server side script to link this page to an RDB variable -->
<%
    if (request['REQUEST_METHOD'] == "POST") {
        tempval = form['sigledEnabled'];
        retval=set_single( 'service.signal.strength.leds.enable='+tempval );
    } else /*GET*/ {
        tempval = get_single( 'service.signal.strength.leds.enable' );
        form['sigledEnabled'] = tempval;
    }
    if(tempval=='1') {
        checked_on="checked"
        checked_off=""
    } else {
        checked_on=""
        checked_off="checked"
    }
%>
```

This usually reads an RDB variable called 'service.signal.strength.leds.enable', whose content is used to set up the variables that control the initial state of the radio button (checked\_on, checked\_off). In the case of a POST (user has hit the 'save' button), the new value is written to the same RDB variable.

```
<body>
<div id="contentWrapper">
<!-- Include the dynamically generated page decorations and menu -->
<% include /menu.html %>
```

This ESP snippet is responsible for creating the header and menu system for the page.

```
<div class="content" >
```



# Utilities

## One Shot Timer Utility

The NetComm Wireless software includes a One Shot Timer Utility (`one_shot_timer`) which effectively provides a programmable countdown timer. When the timer expires it effects a change to an “output” RDB variable. The timer can also be cancelled or reloaded before expiry.

It can optionally provide a feedback “remaining timeout” RDB variable that the caller can poll to calculate how much time is left before the countdown expires. This variable is updated every 100ms.

### Usage

The One Shot Timer Utility is invoked from the command line with a minimum of 2 command line arguments: the input (timer value) and the output RDB variables. All timeout values are in multiples of 10 milliseconds.

For example:

```
rdb_set test.timeout 1000
one_shot_timer -t "test.timeout" -o "test.output"
```

In this example, the One Shot Timer:

1. Reads 1000 from the input (timer value) variable “test.timeout”. This is in tens of milliseconds, so the timeout is 10 seconds.
2. Counts down
3. Sets RDB variable “test.output” to a desired value after 10 seconds when the timer expires. Since the value is not given (via the `-w` command line option), it is set to empty string “” (which is sufficient in triggering templates and the Generalized Event Manager).

### Common Use

The following is an example of how the One Shot Timer Utility might commonly be used:

```
rdb_set test input 1000 # set to 10 seconds
one_shot_timer -t test.input -o test.output -w "expired" -d
```

In 10 seconds when the timer expires, the string “expired” will be written to the “test.output” RDB variable, then the application exits. The “-d” option specifies that `one_shot_timer` runs as a daemon (e.g. the control will be returned to the calling process immediately after `one_shot_timer` was invoked). It is expected that most commonly the output variable will be a trigger for either a template or a Generalized Event Manager trigger variable, e.g. `logic_builder.0.direct_r_trigger`. Therefore, `one_shot_timer` provides a programmable delay capability between inputs and outputs.

### Feedback variable

You can create a feedback variable which can be read by other processes to find out the current state of the countdown.

```
rdb_set test input 1000 # set to 10 seconds
one_shot_timer -t test.input -o test.output -w "expired" -d -f test.feedback
```

The `test.feedback` variable counts down from 1000 to 0 and remains at 0 after the application has exited.

## Multiple instances

While it is completely normal to run multiple instances of `one_shot_timer` to watch different input variables, it's generally not a good idea to run multiple instances of it with the same set of input and output variables.

For example:

```
rdb_set test.input "1000"
one_shot_timer -t test.input -o test.output -w "expired" -d
one_shot_timer -t test.input -o test.output -w "expired" -d
one_shot_timer -t test.input -o test.output -w "expired" -d
```

This will result in three `one_shot_timer` processes running and doing exactly the same thing. To guarantee that only one “`one_shot_timer`” process with the same input is run (and not others are already running when you call the `one_shot_timer`), do the following:

```
rdb_set test.input 0 # will stop any one shot timer processes watching test.input
rdb_set test.input 1000 # set the desired timeout
one_shot_timer -t test.input -o test.output -w "expired" -d # launch one process
```

## Absolute time (-a) option

It is possible, instead of specifying `-t` and the name of an RDB variable, to specify the absolute time value in tens of milliseconds. For example:

```
one_shot_timer -a 100 -o test.output -w "expired" -d -f test.feedback
```

Short of kill, it is not possible to stop a timer started using this method, neither is it possible to change the expiry timeout once the timer application is running. In other words it is not much different to:

```
sleep(1); rdb set test.output "expired"
```

## Arbitrary executable (-x) option

It is possible, instead of specifying `-o` and `-w` options, to provide a command to be executed when the timer expires. For example:

```
one_shot_timer -t test.input -x "shut_down_power 666" -d -f test.feedback
```

`shut_down_power 666` will be executed when/if the timer expires. The new process is always forked by the `one_shot_timer` before the parent `one_shot_timer` process exits.

## Implementing a periodic timer

It is possible to use the `-x` option to start another instance of `one_shot_timer`, thus implementing a periodic, self-restarting timer/process. To do this, create the following shell script (called `periodic.sh`):

```
#!/bin/sh
one_shot_timer -t test.input -x "./periodic.sh" -d -f test.feedback
# your actions, for example, blink LED, etc
```

`test.input` should NOT be set in the script but before the script is first invoked, e.g.:

```
rdb_set test.input 1000
./periodic.sh
```

Please note:

- There will always be only one “`one_shot_timer`” process.
- It can be stopped by writing a 0 to `test.input`.
- You can vary the period at any time by writing a desired timeout to `test.input`.
- The optional feedback variable will count down to 0, then become equal to the `test.input` value again, then count down again and so on, ad infinitum.

## Other usage considerations

- Option -d will daemonize the application e.g. the control will return to the caller.
- if -w (Write value) is not given at all, an empty value will be written on expiry to the rdb output variable.
- The name of input and feedback RDB variables cannot be the same.
- The maximum timeout is 24 hours. Therefore, the maximum value for the rdb timeout variable (given in 10's of milliseconds) is  $8640000 = (24*60*60*100)$
- Each one\_shot\_timer becomes a process in Linux. No measurements are done in regards to footprint of these processes and no recommendations can be made at this point as to how many simultaneous instances can be run without unduly loading the system. It is however, expected to be very light weight.

## System control script

This is a script which is generated at build time and contains the knowledge of how to change the system's operational state. This usually involves GPIOs, but other interfaces may be used, depending on the platform. The sys script supports digital/analogue I/O control commands that are platform independent. These commands are internally converted to GPIO setting commands – NAMUR (input), analogue input, digital input and digital output. Reading external digital input, writing external digital output, reading ADC commands are also centralised as hardware independent commands in this script.

```
root:~# sys
```

```
System control script - knows how to power cycle hardware, configure
audio modes, etc. Functions unavailable on platform are no-ops.
This variant is configured for Bovine/ntc_nw112.
```

```
Usage: sys [-u 0|1] [-otg id|h|d] [-m 0|1] [-w 0|1] [-s 0|1] [-c MODE] [-R] [-B] [-S] [-z
0|1|b]
```

```
-u 0|1          = USB control, disable/enable USB infrastructure
-otg h|d        = Switch USB OTG mode between host and device
-otg id         = Return state of USB OTG id pin - 0 = host
-m 0|1          = WAN module control
-w 0|1          = Wifi control
-s 0|1          = SLIC control
-c MODE         = CPLD audio routing mode (loop, voice, buf)
-r MODE         = RS-232/422/485 mode (off, loop, rs232, rs422, rs485)
-R             = Outputs reset reason (powerup, button, etc.)
-B             = Outputs bootmode button state (normal, alternate)
-S             = Returns sim present status (0=yes, 1=no)
-z 0|1|b       = ZigBee control (b to start ZigBee bootloader)
```

For specific examples of how to use the system control script, see the Serial Port and USB sections later in this guide.

# Inputs/Outputs

(MachineLink 3G Plus only)

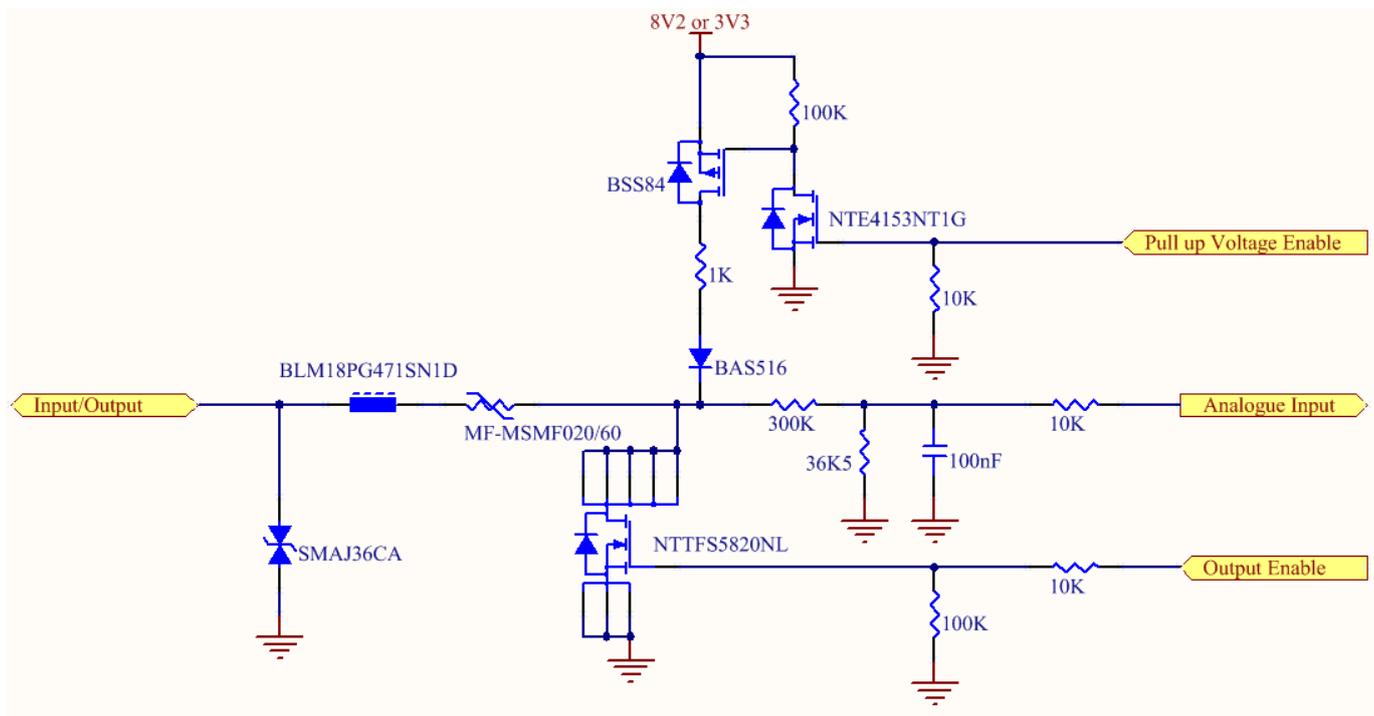
## Overview

The Vodafone MachineLink 3G Plus is equipped with a 6-way terminal block connector providing 3 identical multipurpose inputs and outputs as well as a dedicated ignition input. These inputs and outputs may be independently configured for various functions, including:

- NAMUR (EN 60947-5-6 / IEC 60947-5-6) compatible sensor input
- Proximity sensor input for use with contact closure (open/closed) type of sensors (PIR sensors, door/window sensors for security applications) with the input tamper detection possible (four states detected: open, closed, short and break) by the use of external resistors
- Analogue 0V to 30V input
- Digital input (the I/O voltage measured by the Analogue input and the software making a decision about the input state) with the threshold levels configurable in software
- Open collector output.

## Hardware Interface

The interface of the 3 multipurpose inputs/outputs are based on the circuit diagram below



The **Input/Output** label is the physical connection to the outside world. There are protection devices and resistor dividers to condition the signal prior to it going into the processor. The three labels to the right are the interface to the processor. **Output Enable** activates the Transistor which provides an open collector (ground) output and can sink 200mA at 23°C. It is protected by a resettable fuse and transient protection diode. If used with the pull up resistor, which can be activated by the **Pull up Voltage Enable** pin, then you can have a High or Low output rather than open drain. The resistor can be pulled up to 3V3 for Cmos compatible output or 8.2V by software. The **Analogue Input** pin can read values from 0V to 30V. It is divided by a resistor network to read appropriate levels in the processor. Depending on the sensor type used, the pull up resistor can be switched on or off. If using the NAMUR sensor configuration the pull up will be activated to 8V2 by default.

## Wiring Examples

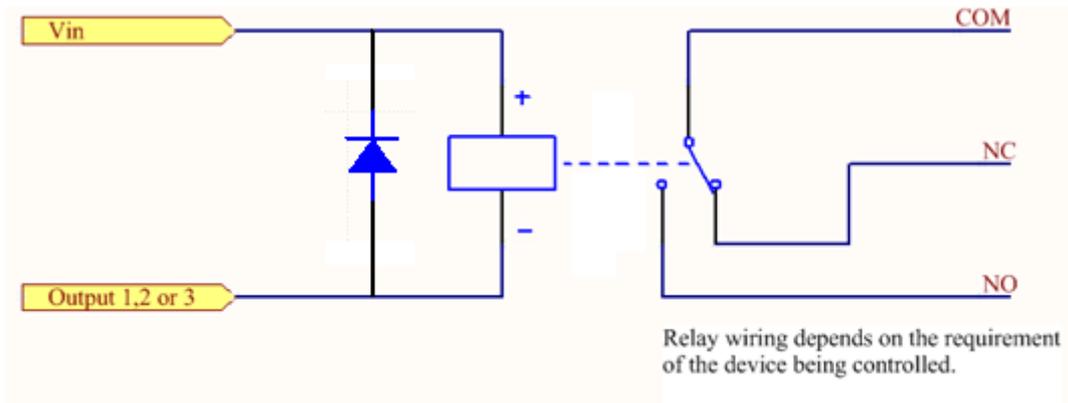
The following examples are shown as a guide as to what can be achieved by the I/O features. It is up to the system integrator to have enough knowledge about the interface to be able to achieve the required results.



Note: NetComm Wireless does not offer any further advice on the external wiring requirements or wiring to particular sensors, and will not be responsible for any damage to the unit or any other device used in conjunction with it. Using outputs to control high voltage equipment can be dangerous. The integrator must be a qualified electrician if dealing with mains voltages controlled by this unit.

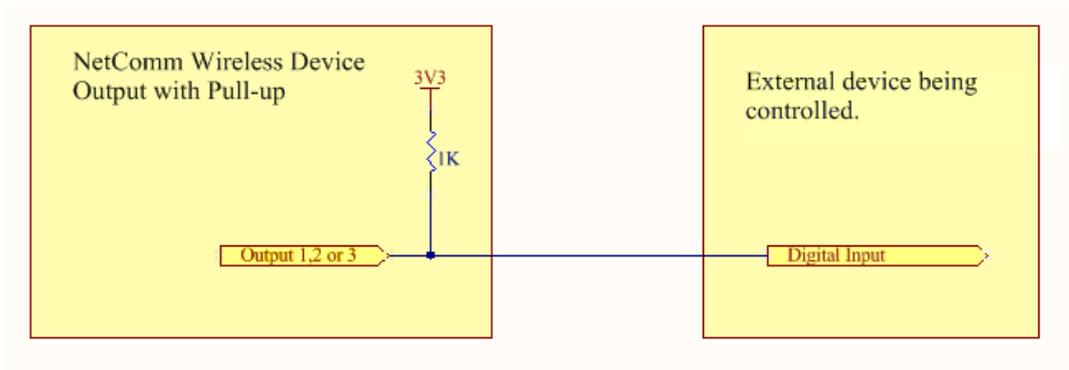
### Open Collector Output driving a relay

Any output can be configured to control a relay. This is an example where the transistor will supply the ground terminal of the solenoid. External voltage is supplied to the other side of the solenoid.



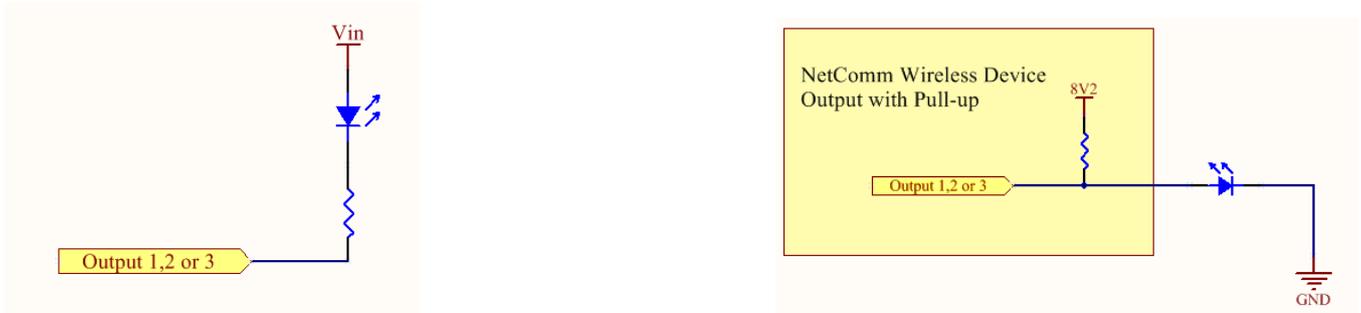
### Logic level Output

An output can be used with the pull up resistor to provide a logic level output which would be suitable to control an external digital device.



### LED Output

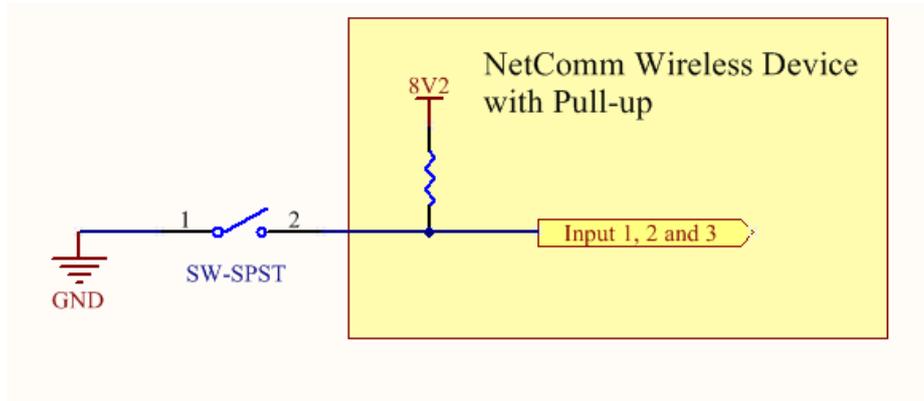
An LED can be controlled by simply providing an open collector ground to an externally powered LED Resistor value and Voltage will need to suit the LED type used. Alternatively an LED can be powered using 8V2 via 1K resistor. The suitability of the LED will need to be investigated.



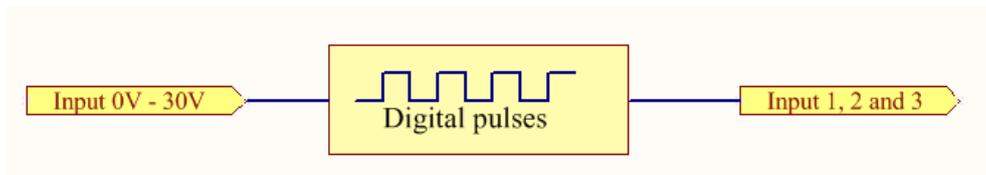
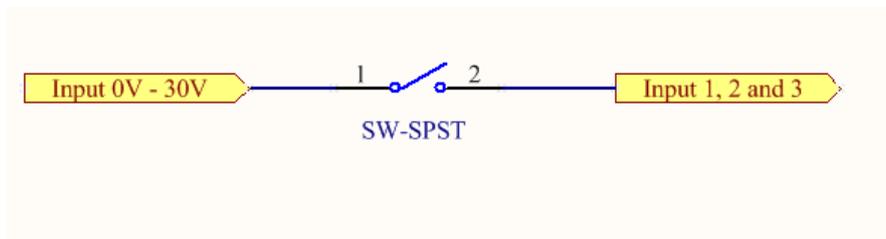
### Digital inputs

There are several ways to connect a digital input. A digital input can be anything from a simple switch to a digital waveform or pulses. The unit will read the voltage in as an analogue input and the software will decode it in a certain way depending on your configuration.

Below is a contact closure type input, which is detecting an Earth. Pull up is activated for this to work.

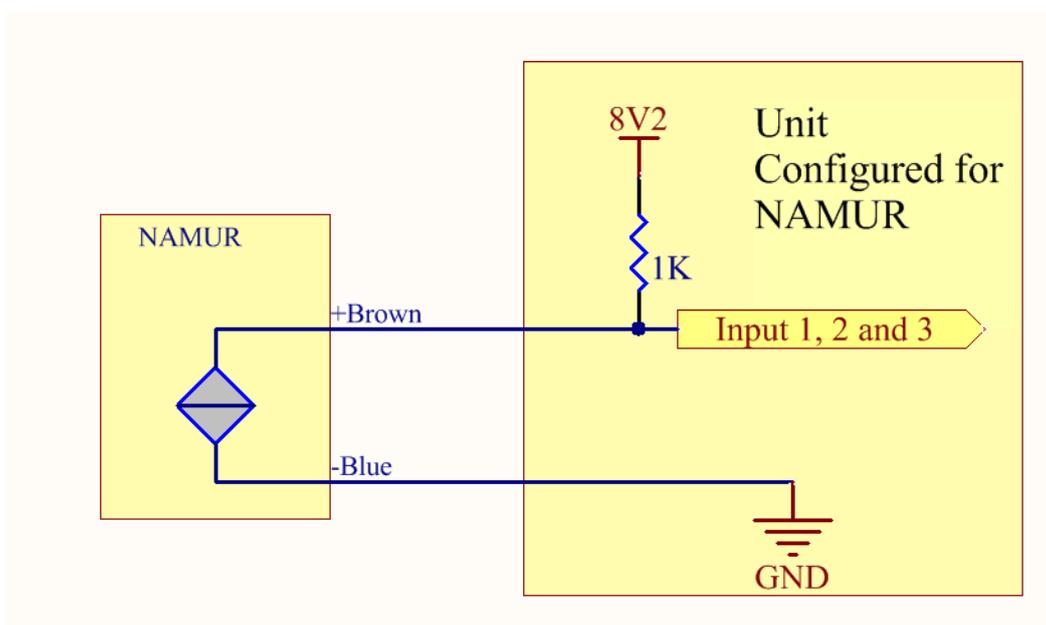


The following input detects an input going high. The turn on/off threshold can be set in the software.



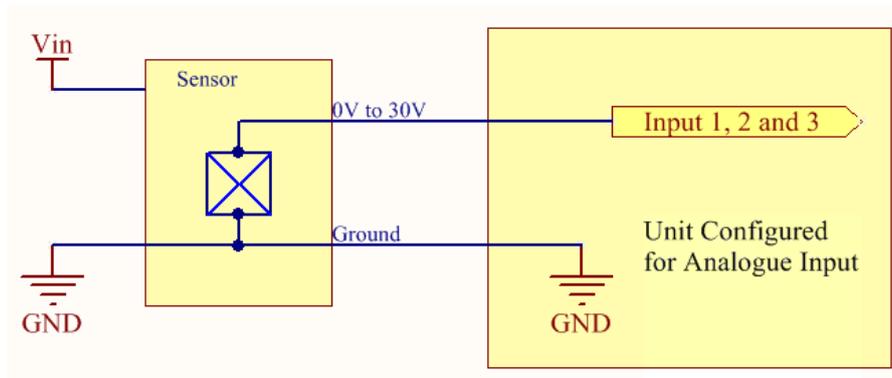
### NAMUR Sensor

A NAMUR sensor is a range of sensors which conform to the EN 60947-5-6 / IEC 60947-5-6 standards. They basically have two states which are reflected by the amount of current running through a sense resistor.



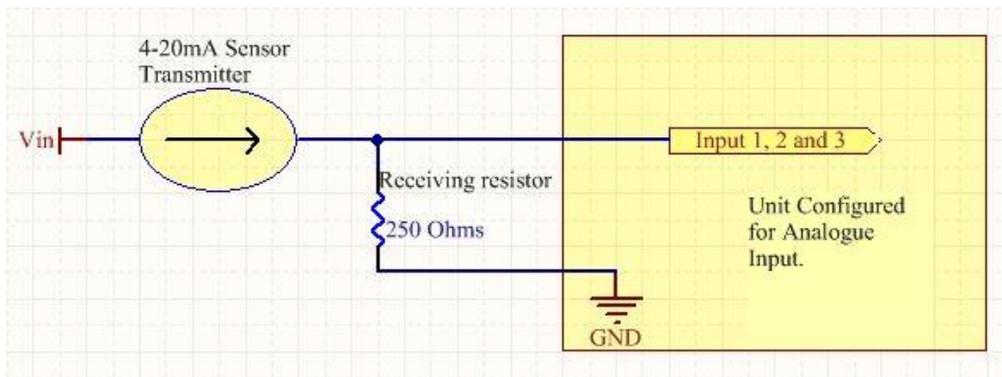
### Analogue Sensor with Voltage output

There are various analogue sensors that connect directly to the unit which can provide a voltage output. These would require an external power source which may or may not be the same as the unit itself. The voltage range they provide can be between 0V and 30V. Some common sensor output ranges include 0V to 10V. These would work on the unit, The pull up resistor is not activated in this case.



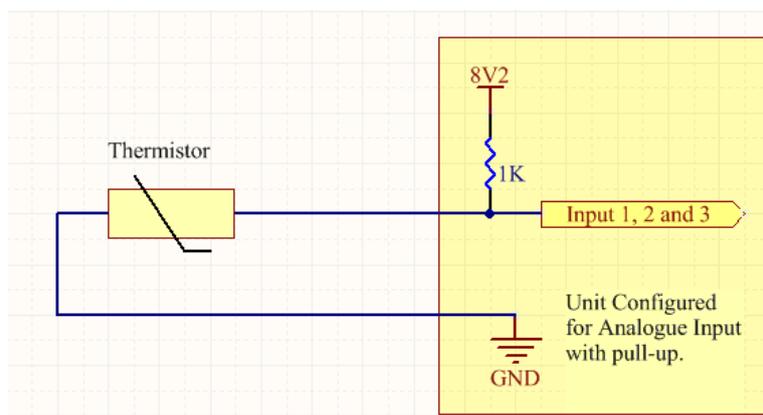
### Analogue Sensor with 4 to 20mA output

Another common type of sensor type is the 4-20mA current loop sensor. It provides a known current through a fixed resistor, usually 250 ohms thus producing a voltage of 0v to 5V at the input. The sensor would require an external power source which may or may not be the same as the unit itself. It will also require an external resistor. The internal pull up resistor is not activated.



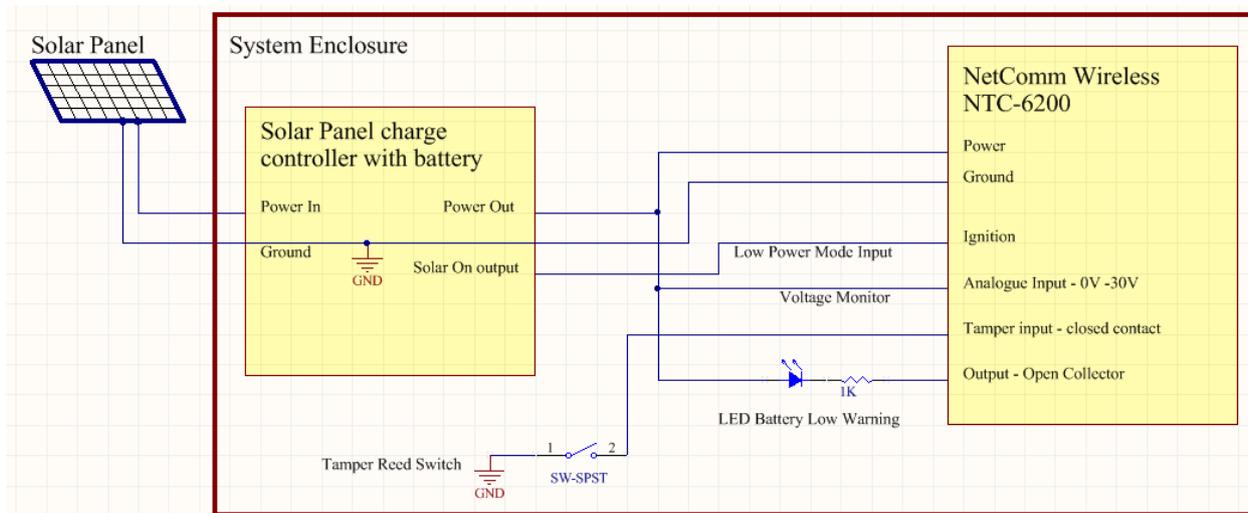
### Analogue Sensor with Thermistor

Some sensors work by changing resistance due to a change, such as temperature, light etc. These may be wired up to an external or internal power source and the resistance can be read into the analogue signal. This will require some software calibration like scaling or offset to map the voltage received to the sensor resistor value. An example below shows the internal pull-up voltage and 1K resistor activated. The voltage received depends on the combination of resistors and the value of the resistance of the sensor itself.



## System Example –Solar powered Router with battery backup

The previous examples of wiring can be used to come up with a system. The following test case is an example of how the I/O's can be used to enhance a simple router setup.



## I/O Design

### Kernel Industrial I/O Subsystem

The NetComm Wireless I/O subsystem relies on one of the newest Linux subsystems called the Industrial I/O (IIO) Subsystem because the Linux hwmon driver is more appropriate for fan speed sensors, temperature sensors and voltage sensors. The Linux IIO subsystem also contains a Linux standard application programming interface consisting mostly of I/O control based messages such as Basic device registration and handling, Event chrdevs, Hardware buffer support and Trigger and software buffer support. These standard ways of accessing ADC will allow the I/O daemon to be a generic application that does not depend on a hardware platform. This structure is part of the Linux Kernel and not proprietary to NetComm Wireless.

### I/O Daemon

The I/O daemon is a main proprietary structure of the NetComm Wireless IO subsystem. It reads raw ADC data from the standard IIO /sys file system to maintain RDB variables that represent ADC values. Additional configurable pre-processing is performed between raw ADC values and RDB ADC values such as noise filtering, down-sampling, multiplying based on internal/external hardware resistor configuration and error correcting (tuning) per ADC channel. These configurable pre-processing settings exist in RDB, preconfigured in each platform based on V\_ADCMAP and changeable in run-time on demand.

### Data flow of sensing information



## Configuring the I/O pins

There are three methods of accessing the I/O pins; the Sys script, the io\_ctl script and the RDB. Each method provides the same set of functionality. The following sections describe how to use each method to configure the I/O pins for your own purposes.

### Using the io\_ctl script

The io\_ctl script performs basic configuration and allows I/O control via the RDB interface. Entering the io\_ctl command at the command line prompt displays the details of its usage.

```

root:~# io_ctl

io_ctl controls Aux IO via RDB interface of io_mgr

usage>
    io_ctl <command> [options...]

command syntax>

* Global IO information commands
    stat                : check to see if daemon is running and reacting
    list                : list all IOs
    set_pull_up_voltage <3.3|8.2> : set pull up voltage
    get_pull_up_voltage : get pull up voltage
    check <IO name>     : check if the IO exists (return code only)

* Per-channel commands

    # Analogue IO commands
    read_analogue <IO name>      : read analogue input
    write_analogue <IO name> <value> : write analogue output

    # Digital IO commands
    read_digital <IO name>      : read digital Input
    write_digital <IO name> <value> : write digital output

    get_cap <IO name>          : get physical capability of IO
    get_mode <IO name>        : get IO mode
    set_mode <IO name>        : set IO mode

    set_pull_up <IO name> <0|1> : set pull up status
    get_pull_up <IO name>      : get current status of pull up
    check_pull_up <IO name>    : check to see if pull up control exists

    set_din_threshold <IO name> <value> : set virtual digital input threshold
    get_din_threshold <IO name>        : get virtual digital input threshold

    set_hardware_gain <IO name> <value> : set hardware gain
    get_hardware_gain <IO name>        : get hardware gain

* High frequency interface commands

    hfi_enable_output : enable high frequency output interface (port is 30000)
    hfi_disable_output : disable high frequency output interface
    hfi_netcat_output  : netcat on the high frequency output interface

    hfi_enable_input : enable high frequency output interface (port is
30001)
    hfi_disable_input : disable high frequency output interface
    hfi_netcat_input <input file> : netcat <input file> to the high frequency input
interface

* Misc.
    help : print this help screen

root:~#

```

## Listing the IOs of the device

It's useful to know before beginning any configuration what IOs you can work with. Use the following command to output a list of all configurable IOs of the device:

```
root:~# io_ctl list
3v8poe
ign
vin
aux1
aux2
aux3
root:~#
```

The table below lists each of the I/Os with a description.

I/O	DESCRIPTION
3v8poe	The onboard 3.8V PoE input.
ign	The external ignition input.
vin	The onboard voltage input.
aux1	The 3 external auxiliary input and output pins.
aux2	
aux3	

The I/Os that will be of most interest to many users are the external I/O pins on the power terminal connector of some models of NetComm Wireless routers. These I/O pins are labelled on the terminal connector as I/O 1, 2 or 3. Through the command line interface, these pins are respectively known as aux1, aux2 and aux3.

## Getting the IO capabilities and setting an IO mode

The `io_ctl` script can be used to return the capabilities of a specified IO, for example:

```
root:~# io_ctl get_cap aux2
digital_output|analogue_input|virtual_digital_input
root:~#
```

We now know that auxiliary input 2 can be used for digital output, analogue input and virtual digital input. Let's set auxiliary input 2 to digital output.

```
root:~# io_ctl set_mode aux2 digital_output
root:~#
```

## Setting pull up status

Before setting the pull up status of an IO, check that pull up control can be performed:

```
root:~# io_ctl check_pull_up aux2
ok
```

Now turn pull up on for aux1:

```
root:~# io_ctl set_pull_up aux2 1
```

Confirm the pull up status:

```
root:~# io_ctl get_pull_up aux2
1
```

A value of 1 means that pull up is enabled, while a value of 0 means it is disabled.

## Setting pull up voltage

The pull up voltage is a global value and may be set at 3.3V or 8.2V. To set the pull up voltage to 8.2V:

```
root:~# io_ctl set_pull_up_voltage 8.2
```

Now confirm that the voltage has been correctly set:

```
root:~# io_ctl get_pull_up_voltage
8.2
```

## Writing digital outputs

When an IO is configured as a digital output, you can use the following command to set digital output to 0:

```
root:~# io_ctl write_digital xaux2 0
root:~#
```

## Reading digital inputs

When an IO is set to a digital input, you can use the following command to read the digital input:

```
root:~# io_ctl read_digital xaux2
0
```

## Setting a digital input threshold

When the device is receiving a digital input, you can set a voltage threshold which is used to determine when the wire is “low” and when it is “high”. Here’s how to set the digital input threshold of auxiliary input 2 to 3V:

```
root:~# io_ctl set_din_threshold xaux2 3.0
```

To confirm the digital input threshold setting:

```
root:~# io_ctl get_din_threshold xaux2
3.0
root:~#
```

## Reading analogue inputs

Each of the three IO pins is capable of reading 0V to 10V input. When an IO pin is set to analogue input mode, you must disable pull up on that pin in order to get an accurate analogue input reading. The following command reads the analogue input on auxiliary input 2 and prints a voltage value.

```
root:~# io_ctl read_analogue xaux2
0.16
root:~#
```

## Setting hardware gain values

The hardware gain is used to instruct the device to perform division on the voltage which is input to an IO pin. This is because the CPU is expecting 1.8V so some conversion must take place on input voltages.

Hardware gain is turned on by default. To turn it off, set the hardware gain to 1:

```
root:~# io_ctl set_hardware_gain xaux2 1
root:~#
```

## High frequency input and output

As the RDB interface is not suitable for extremely high frequency sampling rates (up to maximum hardware specification), the daemon provides two additional TCP ports - one to [extract IO stream] in real-time and the other one to [inject IO stream] back to IO manager. Each TCP stream service allows only a single client to avoid any potential risk of low performance or time sharing issues. Although these services are passive, the extracting feature can easily be redirected to actively send to a certain Internet machine with socat.

### Stream extracting service

This extracting service is enabled by default and is easily accessible with netcat or applicable with socat. A single client is allowed at a time.

### Stream injecting service

As this stream injecting service overrides all physical inputs, this feature is disabled by default and a single client is allowed at a time. Since this service uses the same format of data that the stream extracting service provides, this service can be useful for many kinds of field test purposes. For instance, for trouble shooting purposes, engineers can apply field data stream to their routers to have a better look to address any application errors.

### High frequency interface data format

This information is written in human-readable format. Currently, <IO type> is ain only.

First, enable high frequency output:

```
root:~# io_ctl hfi_enable_output
```

You can then print high frequency output to the device in real-time on the screen:

```
root:~# io_ctl hfi_netcat_output
vin,ain,0:11.90,11.90,11.90,11.90,11.90,11.89,11.88,11.88,11.90,11.88
3v8poe,ain,0:0.01,0.00,0.01,0.01,0.00,0.01,0.00,0.01,0.00,0.01
iaux1,ain,0:8.47,8.47,8.47,8.47,8.48,8.47,8.47,8.47,8.47
iaux2,ain,0:0.17,0.17,0.17,0.17,0.17,0.17,0.17,0.17,0.17,0.15
iaux3,ain,0:0.15,0.14,0.15,0.14,0.15,0.13,0.14,0.14,0.15,0.13
vin,ain,500:11.87,11.90,11.90,11.89,11.90,11.90,11.90,11.90,11.89,11.90
3v8poe,ain,500:0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.01,0.00,0.01
iaux1,ain,500:8.47,8.47,8.47,8.47,8.47,8.47,8.47,8.47,8.48,8.47
iaux2,ain,500:0.18,0.17,0.17,0.16,0.17,0.17,0.17,0.17,0.16,0.17
iaux3,ain,500:0.14,0.13,0.15,0.14,0.14,0.14,0.15,0.14,0.15,0.14
vin,ain,1000:11.91,11.90,11.90,11.90,11.90,11.90,11.90,11.90,11.88,11.89
3v8poe,ain,1000:0.00,0.00,0.00,0.00,0.00,0.01,0.01,0.01,0.01,0.00
iaux1,ain,1000:8.47,8.47,8.47,8.47,8.47,8.47,8.47,8.47,8.46
iaux2,ain,1000:0.17,0.17,0.17,0.17,0.17,0.16,0.17,0.17,0.16,0.17
iaux3,ain,1000:0.15,0.14,0.14,0.14,0.14,0.13,0.14,0.14,0.14,0.13
```

It is also possible to open a TCP socket to access high frequency interface data. This is achieved through the Data stream manager feature available via the web user interface. For further detail on the Data stream manager, please refer to the device's user guide.

## RDB Variables

### I/O Manager Configuration

RDB VARIABLE	DESCRIPTION	PERMISSIONS	POSSIBLE VALUES
sys.sensors.iocfg.mgr.debug	iomgr debug level	Read and write	[0~7, default:4/LOG_INFO]
sys.sensors.iocfg.mgr.enable	<p>When set to 0, this terminates the IO manager and blocks it from running again. Setting this to 1 removes the block from IO manager running but will not cause IO manager to run. To run IO manager again, you should use the <a href="#">IO manager start up script</a>.</p> <p>WARNING: Disabling the IO manager causes the router to stop monitoring all of the I/O pins including the ignition pin. The router will also cease reporting of the DC input voltage on the Status page.</p> <p>While this RDB variable is used to stop the daemon, it cannot be used to check if IO manager is running. Use the sys.sensors.iocfg.mgr.ready RDB variable to check the status of the IO manager.</p>	Read and write	[1=normal operation   0=terminate and do not run again]
sys.sensors.iocfg.mgr.ready	When the daemon is ready to accept RDB commands, this flag becomes "1"	Read only	[1=ready, 0=not ready]
sys.sensors.iocfg.mgr.watchdog	This RDB variable is provided to check the RDB communication with IO manager. As soon as any value is written to this RDB, IO manager resets the RDB back to "1"	Read and write	[only 1]

### Sensory I/O Configuration

RDB VARIABLE	DESCRIPTION	PERMISSIONS	POSSIBLE VALUES
sys.sensors.iocfg.pull_up_voltage	Global pull up voltage selection	Read and write	[3.3   8.2]

sys.sensors.iocfg.rdb_sampling_freq	Global rdb update frequency	Read and write	[integer number, default:5 Hz]
sys.sensors.iocfg.sampling_freq	Global ADC sampling frequency	Read and write	[integer number, default:10 Hz]

### Extra Information

RDB VARIABLE	DESCRIPTION	PERMISSIONS	POSSIBLE VALUES
sys.sensors.info.powersource	Router power source type information - backward compatibility	Read only	[DCJack   PoE   DCJack+PoE]

### Example I/O List (this varies depending on the capabilities of the device)

RDB VARIABLE	DESCRIPTION
sys.sensors.io.3v8poe	The onboard 3.8V PoE input.
sys.sensors.io.ign	The external ignition input.
sys.sensors.io.vin	The onboard voltage input.
sys.sensors.io.xaux1 sys.sensors.io.xaux2 sys.sensors.io.xaux3	The 3 external auxiliary input and output pins.

### I/O RDB Structure

RDB VARIABLE	DESCRIPTION	PERMISSIONS	POSSIBLEVALUES
<b>OUTPUT</b>			
sys.sensors.io.xaux1.d_out	Digital output	Write only	[0 1]
<b>INPUT</b>			
sys.sensors.io.xaux1.adc	Analogue input	Read only	[real number]
sys.sensors.io.xaux1.adc_hw	ADC register value directly from ADC	Read only	[integer number]
sys.sensors.io.xaux1.adc_raw	Raw analogue voltage. actualy voltage that is asserted to ADC	Read only	[real number]
sys.sensors.io.xaux1.d_in	Digital input. ADC digital input is also updating this	Read only	[0 1]
<b>INFORMATION</b>			
sys.sensors.io.xaux1.cap	Hardware capability of IO pin that the IO pin can be configured for - pipe(!) separated multiple combination of the input or output settings.	Read only	[digital_input   digital_output   analogue_input]
<b>CONFIGURATION – PERSISTENT</b>			
sys.sensors.io.xaux1.mode	Current mode of IO pin	Read and write	[digital_input   virtual_digital_in   digital_output   analogue_input]
sys.sensors.io.xaux1.d_in_threshold	Threshold value for virtual digital input. When [mode] is [virtual_digital_input], [adc] and [d_in] are available. If [adc] is equal and higher than [d_in_threshold], [d_in] becomes 1. Otherwise, [d_in] remains 0. This virtual input mode effectively converts an analogue input	Read and write	[real number]

	only IO pin to a digital input IO pin based on the threshold of [d_in_threshold].		
<b>SCALE CONFIGURATION</b>			
sys.sensors.io.xaux1.scale	Scale value to scale raw value to user-friendly value. pre-configured for each platform or variants	Read and write	[real number, default:preconfigured based on platform]
sys.sensors.io.xaux1.correction	Another scale to correct the value after scaling	Read and write	[real number, default:1]
<b>PER-CHANNEL HARDWARE CONFIGURATION</b>			
sys.sensors.io.xaux1.hardware_gain	Hardware scale configuration	Read and write	[0 = 0~1.8v voltage input, 1 = 0~3.2v voltage input]
sys.sensors.io.xaux1.pull_up_ctl	Hardware pull up control	Read and write	[0=disable pull up, 1=enable pull up]

### Starting and stopping the IO manager daemon

While the IO manager automatically starts during the router's boot up sequence, you can manually start or stop the IO manager daemon using the start/stop script. The daemon's process name is "io\_mgr".

To start the IO manager daemon:

```
/etc/init.d/rc.d/io_mgrd start
```

To stop the IO manager daemon:

```
/etc/init.d/rc.d/io_mgrd stop
```

# Serial Port

## (MachineLink 3G Plus only)

The SDK allows configuration of the serial port on the Vodafone MachineLink 3G Plus router. Configuration is achieved through the use of the System control script. This section describes the configuration options available.

### Disabling modem\_emulator

On most products with a serial port, the default configuration has the modem\_emulator daemon running which may prevent CLI commands from working. Before you can attempt any manual configuration of the serial port, you should first disable the modem\_emulator daemon. The best way of achieving this is to set the “confv250.enable” RDB variable to 0:

```
root:~# rdb_set confv250.enable 0
```

To start it again you can simply set the variable to a value of 1.

```
root:~# rdb_set confv250.enable 1
```

### Changing serial port mode (RS232/422/485)

The serial port mode can be configured via the web user interface’s Data stream manager. Please refer to the router’s user guide for further instructions. Alternatively, the serial port mode can be toggled using the system control script.

The following example shows how to set the serial port to RS-422 mode.

```
root:~# sys -r rs422
sys[-sh ]: (Bovine/ntc_nw112) serial_mode: rs422
sys[-sh ]: (Bovine/ntc_nw112) RS-485/422 (full-duplex)
root:~#
```

### Accessing and configuring the serial port from your application

On the MachineLink 3G Plus router, the serial port’s device path is /dev/ttyAPP4.

### Example applications

From within your application, you can send data out through the serial port using socat. The following example sends a text file out of the serial port in raw mode with a baud rate of 115000bps.

```
cat /etc/version.txt |socat - /dev/ttyAPP4,raw,b115200,echo=0
```

This example sends GPS data in raw mode through the serial port using socat.

```
gpspipe -r |socat - /dev/ttyAPP4,raw,b115200,echo=0
```

# USB Port

(MachineLink 3G Plus only)

## Changing USB OTG mode (device/host)

In general, USB OTG mode is fully automated by several kernel components – HCD/UDC controller module, VBUS regulator and USB ID interrupt by the OTG core. The MachineLink 3G Plus router may behave either as a USB device or USB host based on the USB ID pin or OTG negotiation.

As a USB device, routers appear as a multifunction composite device that has a serial port and an Ethernet port. As a USB host, the NetComm Wireless router can accept major Ethernet devices, USB serial devices and USB storage class devices.

To use the USB functions of the router, USB infrastructure must be enabled. By default it is enabled but you can use the following command to ensure it is enabled:

```
root:~# sys -u 1
root:~#
```

You can then set the USB function to host mode, for example, using this command:

```
root:~# sys -otg h
root:~#
```

Then check the current state of the USB OTG ID pin:

```
root:~# sys -otg id
USB port is in host mode.
root:~#
```

When in host mode, the router supports USB Ethernet, USB Serial and USB storage devices.

### USB Ethernet

Multiple USB Ethernet devices can be used and all of the USB Ethernet devices will be bridged to the local interface.

### USB Serial

Only a single USB serial external port is supported which is managed through the Data stream manager section of the web user interface.

### USB Storage

Multiple USB Storage devices are supported and are automatically mounted and available for SDK users. When connected, USB storage devices are automatically mounted with the following naming convention:

```
/var/mnt/USBdisk[A-Z]
```

The supported file systems are FAT16, FAT32 and NTFS.

# SMS

## Sending an SMS

Sending an SMS can be done with `/usr/bin/sendsms` using the following format:

```
sendsms destination_number text store_option
```

For example, to send an SMS containing the message “test sms” to phone number 0412345678:

```
sendsms “0412345678” “test sms”
```

In the example above, the message to be sent is stored on the SIM card or the memory of the device. To send a message without storing it, use the “DIAG” option:

```
sendsms “0412345678” “test sms” “DIAG”
```

When sending an SMS to an overseas destination, the destination number should begin with the ‘+’ character. The SMS tools application sets the default encoding type to GSM7 and intelligently recognizes when special characters have been entered. If special characters have been entered, the SMS tools application silently changes the encoding type to UCS-2 required to send those special characters. Alternatively, if you wish to enforce an encoding type, change the RDB variable below as required:

```
rdb_set smstools.conf.coding_scheme GSM7
```

OR

```
rdb_set smstools.conf.coding_scheme UCS2
```

If the module supports the IRA character set, the router splits sent messages into multiple messages and sends them with a sequence number which is included in the UDH field. The receiver then concatenates these messages into a single message.

## Receiving an SMS

Configuration of the SMS Tools application is available via the web user interface of the router, allowing you to configure forwarding of messages to other mobile numbers, TCP/UDP addresses, email addresses or processing for Diagnostic/Execute commands. Please refer to the SMS Tools section of your router’s user guide for information on configuration via the web user interface.

### Accessing received SMS messages

When SMS messages are received by the router, they are stored temporarily in a spool folder (`/var/spool/sms/incoming`). They are kept there for a short period after which they are processed by the SMS handler script located in `/usr/bin/sms_handler.sh`. After processing they are stored in `/usr/local/cdcs/sms/inbox`. If you wish to create extra custom processing of inbound SMS messages, you can perform functions on the messages located in `/usr/local/cdcs/sms/inbox`. You can change the location that the messages are stored, if so desired, by editing the `sms_common.cfg` file located in `/usr/etc/sms/`.

### Message format

SMS messages located in the inbox folder use the file naming convention of “`rxmsg_XXXXX_unread`” if the message has not been read or “`rxmsg_XXXXX_read`” when it has been read. The ‘X’ characters represent an index number of the message. The first received message will be index number “00000”, the next one will be “00001” and so on. In the case that a message is deleted, the next incoming message will receive the lowest available index number, therefore SMS index numbers do not necessarily indicate a chronological order of when the messages were received.

The following fields are of significance in an incoming message:

FIELD	DESCRIPTION
From	Displays the mobile number of the sender.
From SMSC	The short message service center number.
Sent	The time and date that the message was sent.
UDH	User Data Header. This is an indicator which includes extended information such as total number of messages and sequence number, mostly used for the purpose of concatenating multiple messages.
GSM7 / UCS2	Displays the encoding scheme of the message.

## How to disable SMS messaging

To disable the router's SMS service completely, modify the "smstools.enable" RDB variable so that it is 0.

Use the following command:

```
rdb_set smstools.enable 0
```

# TR-069

## Using TR-069 with your own set of parameters

When using your own custom application, you may wish to use TR-069 to synchronise its settings between a large number of devices. You can achieve this by editing /etc/tr-069.conf. The tr-069.conf file contains the mapping of TR-069 parameters to RDB variables. You may create your own set of parameters and RDB variables for your custom application and list them in the tr-069.conf file so that they can be used over TR-069.

The tr-069.conf file defines the mapping using the following grammar:

```

Definition => Object | Collection | Default | Param

Object => "object" Name "{" (Definition)* "}" ";"
Collection => "collection" Name Handler "{" (Definition)* "}" ";"
Default => "default" "{" (Definition)* "}" ";"
Param => "param" Name ParamType Notify AccessType Handler ";"

Name => /[_a-zA-Z][_a-zA-Z0-9]*/
Notify => "notify" "(" Number "," Number "," Number ")"
Handler => HandlerType "(" (ValueList)? ")"

ValueList => Value ("," Value)*

ParamType => "string" | "int" | "uint" | "bool" | "datetime" | "base64"
AccessType => "readonly" | "readwrite" | "writeonly"
HandlerType => "none" | "const" | "transient" | "dynamic" | "rdbobj" | "rdbmem" | "rdb"
Value => String | Number

String => /"[^"]+"/
Number => /-?[0-9]+/
    
```

For example:

```
param Enable_SNMP bool notify(0,0,2) readwrite rdb("service.snmp.enable", 1, null, null, null, 0);
```

In the example above, the parameter Enable\_SNMP is defined as a Boolean. The notify field stipulates how the value is updated, with a default value of 0, a minimum value of 0 and a maximum value of 2. A notify value of "0" means that the value is never updated, "1" means passive notification, i.e. it is updated when the next session is established, while "2" means active notification, where a value update triggers a session to be established. The access type is read and write and the handler name i.e. the designated RDB variable, is "service.snmp.enable". The following value of 1 indicates that the variable is persistent across reboots. The next four values represents minimum, maximum, validator and default values.

There are several built-in handler types and a generic "dynamic" handler which allows custom logic for special purposes.

HANDLER	DESCRIPTION	ARGUMENTS
none	The default handler, does nothing at all.	none()
const	Constant read-only parameter handler.	const(value)
transient	Memory-only parameters, non-persistent across restarts and reboots.	transient(default)
rdb	RDB parameters, an automatic TR-069 to RDB variable mapping (like persist, but with validation). Persist is a boolean (1 or 0) which turns on the persist flag in RDB. The min and max arguments are context sensitive validation controls, if validator is null the field type is used to determine their meaning. For numeric types min and max are the limits for the value of the field. For string types, min and max validate the length of the field. Passing null for either argument results in no validation of that particular facet. The built-in special purpose validators are current "IPv4", "IPv4Mask" and "URL". For URL the min and max validate the URL length as for strings, for IPv4 and IPv4Mask the min field may be set to zero which enables accepting empty string to indicate no IP specified, if null or otherwise the field must be a valid dotted quad of the specified type.	rdb(rdbKey, persist, min, max, validator, default)

rdbobj	RDB object instance binding (automatic TR-069 to RDB object key convention mapping). Used on collection (multiobject) paths in association with enclosed rdbmem handler parameters. The className is the RDB object class name. Persist is boolean (1 or 0) which controls instance persistence in RDB. The idSelection controls the allocation of object instance IDs, "manual" is not supported. Deletable is also boolean and controls if DeleteObject is allowed to be called on instances.	rdbobj(className, persist, idSelection, deletable)
rdbmem	RDB object member persisted parameters (automatic TR-069 to RDB object key convention mapping). Used on parameters of objects of rdbobj-handled collection (multiobject) paths. The grandparent must be managed by an rdbobj handler. Typically used in the children of the default handler of the rdbobj collection to control the mapping between rdbobject instance members and instance parameters. The memberName is the RDB object instance member name. The min, max, validator and default are as for "rdb" managed parameters - validation and the default value. Unlike rdb the rdbmem parameters inherit their persistence from the configuration of their rdbobj grandparent.	rdbmem(memberName, min, max, validator, default)
dynamic	Manually-coded handler in lua. Implementations in /usr/lib/tr-069/handlers directory as "<handler>.lua". Can have optional extra arguments.	dynamic(handler, default, ...)

# GPS

## (MachineLink 3G Plus only)

When a GPS unit is first turned on, it needs time to find orbit and clock data for the relevant satellites in order to get a precise fix on its location. The time it takes to retrieve this information is referred to as the Time To First Fix (TTFF). The TTFF value can be further broken down into cold, warm or hot times, the difference between these being the amount of information the GPS begins with when it is turned on. The time it takes to get a fix on its location is affected by factors such as interference and horizon information. Open areas are faster than areas such as canyons or urban areas where tall buildings can interfere with the line-of-sight to the GPS receiver.

To improve the time to a precise fix, some modules include support for Assisted GPS. There are two types of assistance:

- Mobile Station Assisted (MSA) – The router receives some assistance from the A-GPS server, such as almanac information, then acquires signals from visible satellites along with an approximation of the time. This information is passed to the A-GPS server which does the position calculations and reports it back to the router.
- Mobile Station Based (MSB) – The router acquires satellite signal more quickly and calculates its position with information provided by an A-GPS server.

## Sending GPS coordinates to a TCP server

The GPS data can be sent out through the serial port as previously described, but if you would prefer to send GPS coordinates instead of NMEA (raw) data to a TCP server, you could use a simple script to provide GPS details and the IMEI number of the device each time the mobile broadband connection is established. This example requires that the router's module is equipped with a standalone GPS receiver.

The RDB variables containing the values of the coordinates are:

```
sensors.gps.0.standalone.latitude
sensors.gps.0.standalone.latitude_direction
sensors.gps.0.standalone.longitude
sensors.gps.0.standalone.longitude_direction
```

Retrieve the values using the `rdb_get` command:

```
root:~# rdb_get sensors.gps.0.standalone.latitude
3348.429011
root:~# rdb_get sensors.gps.0.standalone.latitude_direction
S
root:~# rdb_get sensors.gps.0.standalone.longitude
15108.933068
root:~# rdb_get sensors.gps.0.standalone.longitude_direction
E
root:~#
```

It is important to note the format of the latitude and longitude values. Latitude is DDMM.MMMM while longitude is DDDMM.MMMM, where D represents degree digits and M represents minute digits. The latitude direction and longitude direction values specify N/S & E/W. If an application requires the latitude and longitude values in a different format than what is presented here, conversion of the coordinates into a more math-friendly format (e.g. +/-DD.DDDDD) must be done either as a Lua script or written in C.

We can send the coordinates and the IMEI number to a TCP server running on 192.168.1.49:1516 with the following command:

```
root:~# echo "My gps coordinates are `rdb_get sensors.gps.0.standalone.latitude`:`rdb_get sensors.gps.0.standalone.longitude` Unit IMEI:`rdb_get wwan.0.imei` `date`" | socat TCP:192.168.1.49:1516 -
```

You can configure this operation to run every time the mobile broadband connection is established and standalone GPS is available. The following script may be added into `/etc/init.d/rc3.d/` to start automatically at boot time.

```
#!/bin/sh

old_3g=''

# Infinite loop
while true; do
    sleep 10;
    cur_3g=`rdb_get link.profile.1.status`
    gps_pos=`rdb_get sensors.gps.0.standalone.valid`
    if [ "$cur_3g" != "up" ]; then old_3g=''; continue; fi
    if [ "$cur_3g" = "$old_3g" ]; then continue; fi
    if [ "$gps_spos" != "valid"]; then continue; fi
    # The socat line goes here
    old_3g = cur_3g;
done
```

## List of GPS-related RDB variables

<< GPS function control >>

sensors.gps.0.enable

<< common data >>

sensors.gps.0.common.altitude  
sensors.gps.0.common.date  
sensors.gps.0.common.height\_of\_geoid  
sensors.gps.0.common.latitude  
sensors.gps.0.common.latitude\_direction  
sensors.gps.0.common.longitude  
sensors.gps.0.common.longitude\_direction  
sensors.gps.0.common.time

<< standalone GPS data >>

sensors.gps.0.standalone.3d\_fix  
sensors.gps.0.standalone.GPGGA  
sensors.gps.0.standalone.GPGSA  
sensors.gps.0.standalone.GPGSV  
sensors.gps.0.standalone.GPRMC  
sensors.gps.0.standalone.GPVTG  
sensors.gps.0.standalone.altitude  
sensors.gps.0.standalone.auto\_selection  
sensors.gps.0.standalone.date  
sensors.gps.0.standalone.fix\_quality  
sensors.gps.0.standalone.ground\_speed\_knots  
sensors.gps.0.standalone.ground\_speed\_kph  
sensors.gps.0.standalone.hdop  
sensors.gps.0.standalone.height\_of\_geoid  
sensors.gps.0.standalone.latitude  
sensors.gps.0.standalone.latitude\_direction  
sensors.gps.0.standalone.longitude  
sensors.gps.0.standalone.longitude\_direction  
sensors.gps.0.standalone.magnetic\_track\_made\_good  
sensors.gps.0.standalone.magnetic\_variation  
sensors.gps.0.standalone.magnetic\_variation\_direction  
sensors.gps.0.standalone.mode  
sensors.gps.0.standalone.number\_of\_satellites  
sensors.gps.0.standalone.number\_of\_satellites\_tracked  
sensors.gps.0.standalone.number\_of\_sentences  
sensors.gps.0.standalone.pdop  
sensors.gps.0.standalone.satellitedata  
sensors.gps.0.standalone.sentence\_number  
sensors.gps.0.standalone.status  
sensors.gps.0.standalone.time  
sensors.gps.0.standalone.track\_angle  
sensors.gps.0.standalone.true\_track\_made\_good  
sensors.gps.0.standalone.valid

sensors.gps.0.standalone.vdop

<< assisted GPS data >>

sensors.gps.0.assisted.3d\_fix  
sensors.gps.0.assisted.HEPE  
sensors.gps.0.assisted.LocUncA  
sensors.gps.0.assisted.LocUncAngle  
sensors.gps.0.assisted.LocUncP  
sensors.gps.0.assisted.date  
sensors.gps.0.assisted.height\_of\_geoid  
sensors.gps.0.assisted.lastvalidtime  
sensors.gps.0.assisted.latitude  
sensors.gps.0.assisted.latitude\_direction  
sensors.gps.0.assisted.longitude  
sensors.gps.0.assisted.longitude\_direction  
sensors.gps.0.assisted.time  
sensors.gps.0.assisted.updateinterval  
sensors.gps.0.assisted.valid

<< GPS One variables >>

sensors.gps.0.gpsone.auto\_update.max\_retry\_count  
sensors.gps.0.gpsone.auto\_update.retried  
sensors.gps.0.gpsone.auto\_update.retry\_delay  
sensors.gps.0.gpsone.auto\_update.update\_period  
sensors.gps.0.gpsone.cap  
sensors.gps.0.gpsone.enable  
sensors.gps.0.gpsone.update\_now  
sensors.gps.0.gpsone.updated  
sensors.gps.0.gpsone.urls  
sensors.gps.0.gpsone.xtra.info.gnss\_time  
sensors.gps.0.gpsone.xtra.info.valid\_time  
sensors.gps.0.gpsone.xtra.updated\_time

<< odometer variables >>

sensors.gps.0.odometer\_control  
sensors.gps.0.odometer\_enable  
sensors.gps.0.odometer\_interval  
sensors.gps.0.odometer\_reset  
sensors.gps.0.odometer\_threshold