

Application Note

Extending Router Functionality



© 2025 Advantech Czech s.r.o. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photography, recording, or any information storage and retrieval system without written consent. Information in this manual is subject to change without notice, and it does not represent a commitment on the part of Advantech.

Advantech Czech s.r.o. shall not be liable for incidental or consequential damages resulting from the furnishing, performance, or use of this manual.

All brand names used in this manual are the registered trademarks of their respective owners. The use of trademarks or other designations in this publication is for reference purposes only and does not constitute an endorsement by the trademark holder.

Used symbols



Danger – Information regarding user safety or potential damage to the router.



Attention – Problems that can arise in specific situations.



Information – Useful tips or information of special interest.

Contents

1. Document Content and Structure	1
I Shell Scripting	3
2. Scripting Fundamentals	4
2.1 What Is a Script	4
2.2 Supported Environments	4
2.3 Writing and Executing a Script	4
2.4 Redirecting Output and Error Streams	5
2.5 Basic Script Example	7
2.6 Scripts in Router GUI	8
2.7 Scheduling Scripts with <code>cron</code>	10
3. Specific Implementation Notes and Examples	11
3.1 Handling Incoming SMS with a Custom Script	11
3.2 Email Configuration Notes	12
3.3 Forward Incoming SMS to Email	12
3.4 Send Email on PPP Connection Established	14
3.5 Configure Mobile WAN via SMS	15
3.6 Send SNMP Trap on PPP Connection Established	17
3.7 Switch Between Ethernet WAN and Mobile WAN	18
3.8 Executing AT Commands on Cellular Module	21
3.9 Schedule an Automatic Daily Reboot	23
3.10 Voltage Drop SMS Alert	24
II Python	26
4. Python on Advantech Routers	27
4.1 Introduction to Python Support	27
4.2 Choosing Your Python Router App	27
4.2.1 Python 3 Router App (Full Version)	27
4.2.2 Python 3 Lite Router App	28
4.2.3 Comparison Summary	28
4.3 Installing Python Router Apps	28
4.3.1 Prerequisites	28
4.3.2 Installation Procedure	29
4.3.3 Verifying the Installation	29
4.4 Accessing and Running Python	29
4.4.1 Interactive Python Shell (REPL)	29
4.4.2 Executing Python Scripts	30
4.4.3 Passing Command-Line Arguments	31
4.4.4 Introduction to Python Scripting	31
4.5 Advanced Features	33
4.5.1 Using <code>pip</code> to Install Third-Party Libraries	33

4.5.2	Using <code>venv</code> for Isolated Virtual Environments	34
4.6	Router-Specific Python Development Notes	35
5.	Practical Python Script Examples	37
5.1	Gathering System Information	37
5.2	Basic Network Reachability Test	40
5.3	Simple Log File Monitoring	43
III	Router Apps	46
6.	Getting Started with Router Apps	47
6.1	What are Router Apps	47
6.2	Overview of Development Approaches	47
6.3	General Development Workflow and Tools	48
7.	Router App Structure	49
7.1	Directory Structures for Applications	49
7.2	Application Packaging	55
7.3	Application Lifecycles	56
8.	Building Router Apps	58
8.1	Overview of Development Tools	58
8.2	Cross-Compiler Toolchains	58
8.3	SDK (Software Development Kit)	59
8.4	Building Your First Compiled Application with the SDK	61
8.5	Core Programming for Compiled Applications	67
8.6	Developing Scripted Router Applications (Python)	69
9.	Summary and Best Practices	74
9.1	Key Development Constraints Recap	74
9.2	Best Practices Recap	74
9.3	Firewall Rules for Router Apps	75
IV	Controlling Router Peripherals	78
10.	Digital Input/Output Interfaces	79
10.1	<code>io</code> Utility	79
10.2	Activate Binary Output via SMS	80
10.3	Send Email on Binary Input Activation	82
10.4	Send SNMP Trap on Binary Input State Change	86
11.	Serial Interfaces	87
11.1	Identifying Serial Interfaces	87
11.2	Command-Line Utilities for Serial Ports	87
11.3	Scripting Serial Communication with the <code>um</code> Python Module	91
12.	USB Interface	96

12.1 Storage Access – USB Flash and SD Card	96
12.2 Mounting a USB Flash Drive Partition	96
12.3 Automount USB Flash Disk	97
12.4 Supported USB Serial Converter Chips	100
12.5 Using an Unsupported Serial Converter Chip	100
13. User LED	102
13.1 led Utility	102
13.2 Check IPsec Connection Status via LED	103
13.3 Indicate OpenVPN Status via LED	105
V Constraints	107
14. S1 Router Programming Considerations	108
14.1 Extending the Read-Only Root Filesystem	108
14.2 Adding JavaScript and CSS to the S1 Web Administration Interface	108
14.3 Changing System Configuration Programmatically on S1 Routers	109
15. Hardware Constraints	110
15.1 Non-volatile Memory	110
15.2 RAM Utilization	110
15.3 CPU Performance Considerations	111
VI Custom Firmware Compilation	112
16. Getting Started with Custom Firmware Compilation	113
16.1 Overview of Custom Firmware for Advantech Routers	113
16.2 Prerequisites and Essential Knowledge	113
16.3 Obtaining Firmware Source Code and Build System Components	113
17. Preparing the Build Environment	114
17.1 Setting Up the Development Host System	114
17.2 Setting Up the Cross-Compilation Toolchain	114
17.3 Configuring the Build System	114
17.4 Understanding the Build Directory Structure	114
18. Building the Custom Firmware Image Components	115
18.1 The Firmware Build Process: Step-by-Step	115
18.2 Building and Integrating Open-Source Components	115
18.3 Customizing the Linux Kernel (If Applicable)	115
18.4 Generating and Locating Firmware Image Files	115
18.5 Troubleshooting Common Build Issues	116
19. Installing and Managing Custom Firmware Components	117
19.1 Methods for Installing Built Firmware Components	117
19.2 Initial Boot-up and System Verification	117
19.3 Important Note on Running Custom Firmware Components	118

19.4 Post-Installation Troubleshooting	119
--	-----

VII Special Router Configuration Options 120

20. Overview of Special Configuration Options 121

20.1 Special Options List	121
20.2 How to Apply Special Options	121
20.3 Configuration by a Script	122

VIII Related Documents 123

List of Figures

1 Router Apps Programming Scheme	47
2 The Default Server Option in the NAT Configuration (for IPv4)	75

List of Tables

1 Comparison of Python Router Apps	28
2 Advantech Cross-Compiler Toolchains	58
3 Advantech Router App SDK (ModulesSDK)	59
4 io Utility Commands	79
5 Common stty Options and Settings.	88
6 Supported portd options based on router's help output.	90
7 led Utility Options	102
8 Characteristics of the <code>/var/data</code> Directory Partition	110
9 Characteristics of the <code>/opt</code> Directory Partition	110
10 RAM Memory Parameters	111
11 CPU Architecture	111
12 Summary of Special Router Configuration Options	121

1. Document Content and Structure

This manual introduces various methods for enhancing your Advantech router's core functionality using both built-in firmware capabilities and custom development approaches. Each Advantech router platform offers a degree of extensibility, allowing users to customize and optimize its behavior for specific applications and use cases. The methods described in this document are generally structured by increasing complexity – from simpler scripting solutions to more advanced application and firmware development – and are grouped into the following main parts or chapters:

1. Part *I Shell Scripting* – Scripts are sequences of commands executed via the router's command-line interface (CLI). This method allows users to automate tasks, manage router operations, and extend functionality without modifying the core system firmware. Shell scripts are often the most flexible and immediately accessible way to add simple custom features, enabling tasks like remote management, handling network events, or controlling basic I/O ports.
2. Part *II Python* – Python support on Advantech routers, typically provided via installable Python Router Apps (Full and Lite versions), offers a powerful high-level programming language for more complex automation, data processing, and custom application logic. It facilitates interaction with router functionalities, development of sophisticated scripts, and management of third-party libraries (with `pip` in the Full Python RA), providing a significant step up in capability from basic shell scripting.
3. Part *III Router Apps* – Router Apps (formerly known as User Modules) are self-contained software packages. They can be developed in languages like C/C++, Python, or shell script and then installed on the router. Router Apps provide a structured and robust way to add significant new features or custom applications that are not available in the default firmware. They are ideal for users who need specific, well-integrated functionality, prefer a plug-and-play solution, or need to distribute custom features reliably.
4. Part *IV Controlling Router Peripherals* – This part describes how to interact with and control the router's various hardware interfaces. It covers utilizing the General Purpose I/O (GPIO) interfaces for monitoring and control, managing devices connected to the USB interface (such as storage media or serial converters), and programming user-configurable LEDs for status indication. Examples and usage of command-line utilities like `io` and `led` for managing these peripherals are detailed. There are also available some bash and python script examples utilizing the peripherals.
5. Part *V Constraints* – Standard shell scripting and general Python usage (outside of Router Apps) are typically not supported on the S1 Router product line due to its enhanced security model which restricts direct shell access and script execution in the main OS. To achieve custom automation or extend functionality on S1 devices, developers must implement a custom Router App. This chapter details S1-specific programming considerations, development workflows, and security implications. (Part *III Router Apps* provides general information on Router App development).

This chapter also discusses important hardware-related constraints and considerations relevant to application development on embedded routers. Topics include managing non-volatile memory (flash storage) effectively, understanding RAM limitations, and considering CPU performance implications to ensure applications run reliably without overloading the router's resources.

6. Part *VI Custom Firmware Compilation* – For highly advanced use cases requiring deep integration or modification of the router's operating system components, compiling custom firmware (or selected parts thereof) offers the most powerful solution. This method allows extensive control over software

behavior and feature integration. It primarily involves understanding and potentially recompiling open-source components of the ICR-OS and requires significant knowledge of Linux, embedded systems, firmware development, cross-compilation toolchains, and thorough testing. Users should be aware of the significant implications regarding warranty and support when running custom firmware.

7. Part *VII Special Router Configuration Options* – This chapter describes advanced configuration parameters and system settings that are not typically accessible or configurable via the standard web GUI. These options allow for fine-tuning of specific router behaviors, underlying system services, or enabling specialized features.

Each of these methods for extending router functionality offers a different level of control, complexity, and development effort. This manual aims to provide the necessary information for users to choose and implement the appropriate method to meet their specific requirements, whether they are beginners automating simple tasks or experienced developers building fully custom solutions.

Part I.

Shell Scripting

2. Scripting Fundamentals

2.1 What Is a Script



For more information on writing shell scripts on Linux systems, see the [Advanced Bash-Scripting Guide](#) or the [GNU Bash Reference Manual](#).

Advantech routers support scripting in a manner similar to general-purpose Linux operating systems. A *script* is essentially a text file containing a sequence of commands that the router's command-line interpreter (shell) can execute. Scripts allow you to automate tasks, manage router operations, and extend functionality without modifying the router's core firmware. They represent the simplest method for adding custom functions.

2.2 Supported Environments

Scripts on Advantech routers are written using a POSIX-compatible shell scripting language. The default shell invoked by `/bin/sh` is typically a version of `ash` (from BusyBox), but `bash` might also be available depending on the router model and firmware version. For most common scripting tasks, the differences between these shells are minimal when adhering to POSIX standards.

When writing scripts, you can utilize the standard set of Linux/Unix utilities provided by the BusyBox environment. To list available built-in commands, you can run `busybox --list` or press the `Tab` key twice in the shell. Additionally, Advantech provides several proprietary utilities specific to the router's hardware and functions (e.g., `gsmsms`, `io`, `led`, `gsmat`, `status`). Both the standard BusyBox commands and Advantech-specific utilities are documented in the [Command Line Interface](#) application note.

2.3 Writing and Executing a Script

File Format

Scripts should be saved as plain text files, typically with a `.sh` extension. It is crucial to use Unix-style line endings (LF only) rather than Windows-style line endings (CRLF), as the latter can cause execution errors on Linux. You can create scripts using a text editor on your PC and then transfer them to the router (e.g., via SCP or SFTP to directories like `/root` or `/var/data`), or create them directly on the router using built-in editors like `vi`, refer to [Busybox vi tutorial](#), or the Here-Document method described below.

Shebang Line

Every script should begin with a *shebang* line, which specifies the interpreter that should execute the script. The standard shebang for POSIX-compatible scripts is: `#!/bin/sh`

Here-Document Method

The here-document syntax provides a convenient way to embed multi-line text (like a script's content) directly within another script or command. This is particularly useful when defining scripts within the router's GUI (see Section 2.6) or when creating script files dynamically from the console.

A here-document redirects lines following a command until a specific delimiter is encountered. The syntax is `command << DELIMITER`, followed by the input lines, and ending with `DELIMITER` on a line by itself.

Making the Script Executable

Before a script file can be run directly, it needs execute permissions. Use the `chmod` command to set these permissions. For example, to give the owner read, write, and execute permissions, and others read and execute permissions: `chmod 755 your_script.sh` Alternatively, more restrictive permissions like `chmod 700 your_script.sh` (owner only) might be appropriate.

Running the Script

To execute a script located in the current directory, use its relative path: `./your_script.sh` If the script is located elsewhere, provide the full path: `/path/to/your_script.sh` Alternatively, you can explicitly invoke the shell to run the script, which doesn't require execute permissions: `sh /path/to/your_script.sh`

2.4 Redirecting Output and Error Streams

Shell scripts produce output on two main streams:

- **Standard Output (stdout, file descriptor 1):** Used for normal program output.
- **Standard Error (stderr, file descriptor 2):** Used for error messages and diagnostics.

By default, both streams are displayed on the console. When running scripts non-interactively or logging their activity, it's often desirable to capture this output into files.

Redirecting stdout

Use the `>` operator to redirect stdout to a file, overwriting the file if it exists:

```
./your_script.sh > /path/to/logfile.log
```

Use `>>` to append stdout to the file:

```
./your_script.sh >> /path/to/logfile.log
```

Redirecting stderr

Use `2>` to redirect stderr to a file:

```
./your_script.sh 2> /path/to/errorfile.log
```

Use `2>>` to append stderr.

Redirecting Both stdout and stderr to the Same File

To capture all output (both normal messages and errors) in a single file, redirect stderr to the same location as stdout using the notation `2>&1`. This must appear *after* the stdout redirection.

Syntax

```
./your_script.sh > /path/to/logfile.log 2>&1 (Overwrite log file)
```

```
./your_script.sh >> /path/to/logfile.log 2>&1 (Append to log file)
```

Order Matters

The order of redirections is critical. `2>&1` redirects stderr to wherever stdout is *currently* pointing.

- Correct (stdout and stderr go to file): `command > file.log 2>&1`
- Incorrect (stderr still goes to console): `command 2>&1 > file.log` (stderr is redirected to the original stdout (console) *before* stdout is redirected to the file).


Discarding Output

To discard output, redirect it to the special null device `/dev/null`.

- Discard stdout: `command > /dev/null`
- Discard stderr: `command 2> /dev/null`
- Discard both: `command > /dev/null 2>&1`

2.5 Basic Script Example

Below is an example script, intended to be saved in a file named `test.sh`, that checks network connectivity to a target host and appends a timestamped result to a log file.



```
#!/bin/sh
# test.sh -- Check network connectivity and log status

# --- Configuration ---
TARGET="8.8.8.8" # IP address or hostname to ping
LOGFILE="/var/log/network_status.log" # Path to the log file
# --- End Configuration ---


# Get current timestamp
TIMESTAMP=$(date +"%Y-%m-%d %H:%M:%S")

# Optional: Exit immediately if any command fails
# set -e

# Ping the target host once (-c 1) with a timeout (e.g., -W 2 for 2 seconds)
# Redirect ping's stdout and stderr to /dev/null as we only care about exit status
if ping -c 1 -W 2 "$TARGET" > /dev/null 2>&1; then
    # If ping succeeds (exit code 0), log UP status
    echo "$TIMESTAMP: Network is UP (reachable $TARGET)" >> "$LOGFILE"
else
    # If ping fails (non-zero exit code), log DOWN status
    echo "$TIMESTAMP: Network is DOWN (cannot reach $TARGET)" >> "$LOGFILE"
fi

# Explicitly exit with success status
exit 0
```

To run this script, first ensure it has execute permissions, then execute it using its path (as described in Section 2.3):



```
chmod +x test.sh # Or chmod 755 test.sh
./test.sh
```

This executes the script once, performing a single connectivity check and appending the result to `$LOGFILE`. Another common approach involves running the script's core logic within an infinite `while true` loop, typically used for continuous monitoring tasks. If the script needs to run periodically at specific intervals or at a precise time, the `cron` scheduling utility can be used (see Section 2.7 *Scheduling Scripts with cron*).

2.6 Scripts in Router GUI


The router's web interface provides dedicated sections for defining custom scripts that run automatically in response to specific system events. Navigate to *Configuration* → *Scripts*, where you will find the following subpages:

- **Startup Script:** Executed once every time the router powers on or after a factory reset. Ideal for initial setup, configuration checks, or launching background monitoring processes.
- **Up/Down IPv4 Scripts:** Contains fields for an "Up" script (executed when the primary WAN IPv4 connection is established) and a "Down" script (executed when the primary WAN IPv4 connection is lost). These scripts receive connection-specific parameters (like interface name, IP address).
- **Up/Down IPv6 Scripts:** Similar to IPv4 scripts, but triggered by the establishment ("Up") or loss ("Down") of the primary WAN IPv6 connection, receiving relevant IPv6 parameters.

For detailed instructions on parameters passed to Up/Down scripts and specific use cases, refer to the Configuration Manual for your router model.

Simple GUI Script Example

Below is a syntax example for defining a simple script directly in the GUI (e.g., Startup Script). This script executes its commands once when the corresponding event occurs.



```
#!/bin/sh

# Define variables
PhoneNumber="+420123456789"
Message="Router event triggered SMS."

# Use the variables
sms "$PhoneNumber" "$Message"

exit 0
```

Creating and Running a Script File via GUI Script (Here-Document)

If you need a script defined in the GUI (like the Startup Script) to create a separate, potentially more complex script file and then execute it (possibly in the background), the Here-Document method (Section 2.3) is recommended, as shown below:

```
#!/bin/sh
# This script (run from GUI, e.g., Startup) creates and executes /tmp/script.sh

# Define path for the script to be created (use RAM disk like /tmp or /var/run)
SCRIPT_PATH="/tmp/script.sh"

# Use cat with a here-document (EOF) to write the script content
# Note: Using 'EOF' (quoted) prevents variable expansion by *this* script.
# Variables like $Num inside the heredoc will be treated literally when written.

cat > "$SCRIPT_PATH" << 'EOF'
#!/bin/sh
# This is the content of the created script (/tmp/script.sh)

# Define variables *within this script*
Num="+420123456789"
MessageBody="Sending SMS from background script..."

# Example action: Send SMS
sms "$Num" "$MessageBody"

# Add more complex logic here...

exit 0
EOF
# End of here-document

# Make the created script executable
chmod +x "$SCRIPT_PATH"

# Execute the created script in the background
"$SCRIPT_PATH" &

exit 0 # Indicate successful completion of the startup task
```



Variable Expansion in Here-Documents

Note the handling of variables (like `$Num`) when using the here-document method:

- `<< EOF` (Delimiter unquoted): Variables inside the here-document (e.g., `$VAR`) are expanded by the outer script (the one containing the `cat` command) *before* the content is written to the file. The value of the variable from the outer script's context is embedded.
- `<< 'EOF'` (Delimiter quoted): The content between `'EOF'` delimiters is treated literally. Variables (e.g., `$VAR`) are written as the literal string `$VAR` into the created file. This is generally the desired behavior if the variable is intended to be defined or used *within* the created script itself, as demonstrated in the example above.
- Escaping the dollar sign (`\$VAR`) when using the unquoted form (`<< EOF`) also prevents expansion by the outer script and achieves the same literal effect as quoting the delimiter, but using `<< 'EOF'` is often considered clearer.

2.7 Scheduling Scripts with cron



For further cron scheduling examples and advanced syntax, see crontab.guru.

The `cron` daemon on ICR-OS allows you to run scripts at specified times or intervals. Jobs are defined in the system crontab file `/etc/crontab`. In this file, each line defines a job and follows the syntax below, where the first five fields specify the schedule:

```
<minute> <hour> <day--of--month> <month> <day--of--week> <user> <command>
```

Schedule Field Definitions

- `*` (asterisk) – matches all valid values (e.g., every minute, every hour).
- `m--n` (range) – matches any value between `m` and `n` (inclusive).
- `*/s` (step) – matches every `s` th value (e.g., `*/5` in the *minute* field means –every 5 minutes–).
- `v1,v2,...` (list) – matches any of the comma-separated values.

Valid value ranges

- Minute: `0--59`
- Hour: `0--23`
- Day of month: `1--31`
- Month: `1--12`
- Day of week: `0--7` (both `0` and `7` = Sunday)

Example Entries

- Every minute: `* * * * * root /path/to/script.sh`
- Every 5 minutes: `*/5 * * * * root /path/to/script.sh`
- Hourly at minute 0: `0 * * * * root /path/to/script.sh`
- Daily at 02:30 AM: `30 2 * * * root /path/to/script.sh`
- Monthly on the 1st at midnight: `0 0 1 * * root /path/to/script.sh`
- Weekly on Mondays at 03:00 AM: `0 3 * * 1 root /path/to/script.sh`

Starting `cron`

After creating or updating `/etc/crontab`, start the daemon by `service cron start` or by `crond &`. To verify it's running you can use `ps | grep cron`.

Persistence Across Reboots

If `/etc/crontab` is reset on router reboot, ensure it is recreated in your Startup Script (Section [2.6 Scripts in Router GUI](#)). For example, the startup script may look like this:

```
#!/bin/sh
# WARNING: This overwrites the entire crontab. See AttentionBox in Sec 2.7.1.
cat << 'EOF' > /etc/crontab
*/15 * * * * root /path/to/script.sh
# Add other required system or application cron jobs here
EOF
crond &
```



3. Specific Implementation Notes and Examples

3.1 Handling Incoming SMS with a Custom Script

This section describes an option for handling incoming SMS messages received by the router using a custom script. This is achieved using a script located at `/var/scripts/sms`.



This file path (`/var/scripts/sms`) resides within the router's volatile RAM filesystem. This means the script file and its contents will be lost upon router reboot or power loss.

To ensure the custom SMS handling logic persists across reboots, the `/var/scripts/sms` script must be recreated each time the router starts. A common method is to use the router's Startup Script functionality. Add a code block similar to the following to the Startup Script configuration page (*Configuration* → *Scripts* → *Startup Script*) to automatically create the `/var/scripts/sms` file at boot:



```
#!/bin/sh
# Create /var/scripts/sms using a here-document in the Startup Script
cat << 'EOF' > /var/scripts/sms
#!/bin/sh
# This is the custom SMS handler script /var/scripts/sms
# Insert your SMS-handling logic below
# Example: Log received SMS details
logger -t sms_handler "SMS received. Authorized: $1, Sender: $2, Text: $3 $4 $5 $6 $7 $8 $9"

# Add custom actions based on sender ($2) and message content ($3-$9) here...

exit 0
EOF
```

When an SMS message is received and this feature is enabled (see InfoBox below), the system executes the `/var/scripts/sms` script, passing the following parameters:

- `$0` – The name of the script being executed (always `sms`), set automatically by the shell.
- `$1` – A flag indicating if the sender's phone number matches one of the numbers configured in the *Phone Number x* fields within the router's SMS settings GUI (`1` for a match, `0` otherwise).
- `$2` – The sender's mobile phone number (MSISDN).
- `$3` ... `$9` – The first seven words (space-separated tokens) extracted from the body of the received SMS message.

Within your `/var/scripts/sms` script, you can use these positional parameters (`$1` through `$9`) to implement custom actions based on the sender and content of the SMS. This allows for creating custom SMS commands, for example, to query router status, toggle interfaces, or trigger specific application logic.

- To enable the execution of the `/var/scripts/sms` script:
 - Cellular connection must be properly configured and enabled under *Configuration* → *Mobile WAN*.
 - The *Enable remote control via SMS* option must be checked under *Configuration* → *Services* → *SMS*.
 - Configuring specific *Phone Number x* entries on *Services* → *SMS* page is optional for enabling the script itself but affects the value of the `$1` parameter.
- **Precedence Rule:** If a *Phone Number x* is configured and matches the sender's number, and the received SMS message matches the format of one of the router's built-in SMS control commands (e.g., `get ip`, `reboot`), the built-in command will be executed instead of the custom `/var/scripts/sms` script. The custom script is bypassed in this scenario.
- The `/var/scripts/sms` file generally does not require execute permissions.
- If you need to determine the phone number (MSISDN) of the SIM card currently installed in the router, you can find it by sending an SMS message from the router using the *Administration* → *Send SMS* page to another phone.

3.2 Email Configuration Notes

To send emails from your router, you need to configure an SMTP server at *Configuration* → *Services* → *SMTP*. Ensure the email service is accessible and functional; you may need to properly configure the firewall and NAT settings to allow outgoing SMTP traffic.

You can send a test email from the console by issuing this command:

```
email -t address@domain.ext -s "Test from router" -m "Just testing email functionality."
```


3.3 Forward Incoming SMS to Email

This script relies on the custom SMS handling mechanism (Chapter [3.1 Handling Incoming SMS with a Custom Script](#)) and requires a correctly configured SMTP server (*Configuration* → *Services* → *SMTP*).

This example script uses the custom SMS handler (`/var/scripts/sms`) to automatically forward details of every incoming SMS message to a specified email address.

Startup Script

This script creates the `/var/scripts/sms` handler script in RAM at boot time.



```
#!/bin/sh


# Create the SMS handler script in RAM
cat > /var/scripts/sms << 'EOF'
#!/bin/sh

# Specify email address
EMAIL="john.doe@email.com"

# Forward incoming SMS via email
email -t "$EMAIL" \
      -s "Received SMS from $2" \
      -m "Authorized: $1, Text: $3 $4 $5 $6 $7 $8 $9"
EOF
```

How It Works

- The Startup Script uses `cat > ... << EOF` to create the handler script `/var/scripts/sms`.
- Inside the handler script:
 - `EMAIL="john.doe@email.com"` : Defines the recipient email address.
 - The `email` utility is called to send the message.
 - `-t "$EMAIL"` : Sets the recipient address.
 - `-s "Received SMS from $2"` : Sets the subject line, including the sender's phone number (`$2`).
 - `-m "Authorized: $1, Text: $3 $4 $5 $6 $7 $8 $9"` : Sets the message body, including:
 - * The authorization flag (`$1`): `1` if the sender is in the router's authorized list, `0` otherwise.
 - * The sender's phone number (`$2`) is included in the subject.
 - * The first seven words of the SMS text (`$3` through `$9`).
- This script forwards details of every incoming SMS, regardless of sender or content, to the specified email address.



The standard SMS handler mechanism might only provide the first 7 words of the SMS text (`$3` through `$9`). Longer messages may be truncated in the forwarded email.

3.4 Send Email on PPP Connection Established



Make sure you have correctly configured the SMTP server in *Configuration* → *Services* → *SMTP*. Refer to Section [3.2 Email Configuration Notes](#).

This script sends an informational email when a PPP (Point-to-Point Protocol) connection for the WAN interface is established. It can be related to IPv4 or IPv6 addressing, depending on where the script is placed in the router's GUI: either in the *Up script IPv4* field or the *Up script IPv6* field under *Configuration* → *Scripts*.

IPv4 Up Script

This script should be placed in the *Up script IPv4* field.

```
#!/bin/sh

# Specify email address
# Specify email address
EMAIL=john.doe@email.com

# Send email using parameters passed by the system
# $1 = interface name (e.g., ppp0)
# $4 = assigned IPv4 address
email -t $EMAIL \
-s "Router has established IPv4 PPP connection." \
-m "Interface: $1; IP address: $4"
```

How It Works

- The script defines the destination email address in the `EMAIL` variable.
- When the PPP interface establishes an IPv4 connection, the router executes this script, passing several parameters. This script uses:
 - `$1` – the name of the network interface (e.g., `ppp0`).
 - `$4` – the IP address assigned to the interface.
- The `email` utility (typically located at `/usr/bin/email`) is called to send the notification:
 - `-t $EMAIL` - specifies the recipient's email address.
 - `-s "..."` - sets the subject line of the email, including the interface name.
 - `-m "..."` - sets the message body, including the interface name and assigned IP address passed as parameters.

An example of a received email based on the script above:

Subject: Router has established IPv4 PPP connection on ppp0
 Content: Interface: ppp0; IP address: 192.0.2.100

3.5 Configure Mobile WAN via SMS



This script relies on the custom SMS handling mechanism described in Section [3.1 Handling Incoming SMS with a Custom Script](#). Ensure that mechanism is active and properly configured.

This example demonstrates the implementation of a new SMS command, which can set the *Network type* and the *Default SIM card* for the mobile WAN configuration. The command which should be sent by SMS to the router has the following syntax:

```
PPP <NetworkType> <DefaultSIM>
```

Where:

- **<NetworkType>** specifies the desired network technology. Valid options (depending on router model support) are:
 - **AUTO** : Switches *Network Type* of both SIM cards to *automatic selection*.
 - **GPRS** : Switches *Network Type* of both SIM cards to *GPRS/EDGE*.
 - **UMTS** : Switches *Network Type* of both SIM cards to *UMTS/HSPA*.
 - **LTE** : Switches *Network Type* of both SIM cards to *LTE*.
 - **NR5G** : Switches *Network Type* of both SIM cards to *NR5G*.
- **<DefaultSIM>** specifies the default SIM card:
 - If equal to **1** , sets SIM 1 as default.
 - If equal to **2** , sets SIM 2 as default.

Example SMS: `PPP LTE 1` (Sets network type to LTE for both SIMs and makes SIM 1 the default).

How It Works (script on next page)

- The startup script (shown on the next page) creates the `/var/scripts/sms` handler script in the router's RAM filesystem using a here-document (`cat > ... << EOF`). This ensures the handler exists after a reboot.
- The handler script (`/var/scripts/sms`) is executed by the system when an SMS is received.
- It first checks if the sender is authorized by verifying if the first parameter `$1` is equal to `1` .
- If authorized, it checks if the third word of the SMS (`$3`) is "PPP" (case-sensitive in the example script).
- If the command word matches, the script processes the fourth word (`$4` , network type) and fifth word (`$5` , default SIM).
- **NetworkType:** Based on the value of `$4` ("AUTO", "GPRS", etc.), a series of `if/elif` statements selects the appropriate `sed` command. This command modifies the `/etc/settings.ppp` file in place (`-i`), updating the `PPP_NETTYPE=` and `PPP_NETTYPE2=` lines with the corresponding numeric code (0 for AUTO, 1 for GPRS, 2 for UMTS, 3 for LTE, 8 for NR5G).
- **DefaultSIM:** Similarly, if `$5` is "1" or "2", another `sed` command updates the `PPP_DEFAULT_SIM=` and `PPP_BACKUP_SIM=` lines in `/etc/settings.ppp` .
- **Reboot:** After potentially modifying the settings file, the script immediately executes the `reboot` command to apply the changes.

Startup Script

```
#!/bin/sh

# Create the SMS handler script in RAM
cat > /var/scripts/sms << EOF
#!/bin/sh
if [ "$1" = "1" ]; then
    if [ "$3" = "PPP" ]; then
        if [ "$4" = "AUTO" ]; then
            sed -e "s/(PPP_NETTYPE=\\).*/10/" \
                -e "s/(PPP_NETTYPE2=\\).*/10/" \
                -i /etc/settings.ppp
        elif [ "$4" = "GPRS" ]; then
            sed -e "s/(PPP_NETTYPE=\\).*/11/" \
                -e "s/(PPP_NETTYPE2=\\).*/11/" \
                -i /etc/settings.ppp
        elif [ "$4" = "UMTS" ]; then
            sed -e "s/(PPP_NETTYPE=\\).*/12/" \
                -e "s/(PPP_NETTYPE2=\\).*/12/" \
                -i /etc/settings.ppp
        elif [ "$4" = "LTE" ]; then
            sed -e "s/(PPP_NETTYPE=\\).*/13/" \
                -e "s/(PPP_NETTYPE2=\\).*/13/" \
                -i /etc/settings.ppp
        elif [ "$4" = "NR5G" ]; then
            sed -e "s/(PPP_NETTYPE=\\).*/18/" \
                -e "s/(PPP_NETTYPE2=\\).*/18/" \
                -i /etc/settings.ppp
        fi

        if [ "$5" = "1" ]; then
            sed -e "s/(PPP_DEFAULT_SIM=\\).*/11/" \
                -e "s/(PPP_BACKUP_SIM=\\).*/12/" \
                -i /etc/settings.ppp
        elif [ "$5" = "2" ]; then
            sed -e "s/(PPP_DEFAULT_SIM=\\).*/12/" \
                -e "s/(PPP_BACKUP_SIM=\\).*/11/" \
                -i /etc/settings.ppp
        fi

        reboot
    fi
fi
EOF
```



3.6 Send SNMP Trap on PPP Connection Established



Make sure you have correctly configured the SNMP manager in *Configuration* → *Services* → *SNMP*.

This script sends an SNMP trap to the configured SNMP manager when the PPP (Point-to-Point Protocol) connection for the WAN interface is established. It can be related to IPv4 or IPv6 addressing, depending on where the script is placed in the router's GUI (*Up script IPv4* or *Up script IPv6*).

IPv4 Up Script

Place this script in the *Up script IPv4* field.



```
#!/bin/sh

# Specify SNMP manager address
SNMP_MANAGER=192.168.1.2


snmptrap -g 3 $SNMP_MANAGER
```

How It Works

- The script defines the destination IP address of the SNMP manager in the `SNMP_MANAGER` variable.
- When the PPP interface establishes an IPv4 connection, the router executes this script.
- The `snmptrap` utility sends a generic SNMP trap message:
 - `-g 3` specifies sending a standard generic trap of type `linkUp` (numeric value 3). This indicates that a network interface has come up.
 - `$SNMP_MANAGER` provides the IP address where the trap should be sent.
- The configured SNMP manager receives this generic `linkUp` trap and can process it based on its rules. The trap itself, using only `-g 3`, does not contain specific information about which interface came up or its IP address. More complex `snmptrap` commands would be needed to include such details as variable bindings.

3.7 Switch Between Ethernet WAN and Mobile WAN

This script enables automatic switching between a primary Ethernet WAN connection (assumed to be `eth1`) and a backup Mobile WAN (PPP) connection. The PPP connection becomes active when ping tests to a defined IP address fail, indicating that the primary WAN connection is unavailable. When the primary WAN recovers, it switches back.

 This script makes assumptions about network interfaces, configuration files (`/etc/settings.eth`), and requires careful adaptation to your specific network setup and router model. Directly manipulating routes and firewall rules requires caution.

Startup Script

```
#!/bin/sh

# Specify IP addresses
WAN_PING=192.168.2.1
WAN_GATEWAY=192.168.2.1
WAN_DNS=192.168.2.1

. /etc/settings.eth

/sbin/route add $WAN_PING gw $WAN_GATEWAY
/sbin/iptables -t nat -A PREROUTING -i eth1 -j napt
/sbin/iptables -t nat -A POSTROUTING -o eth1 -p ! esp -j MASQUERADE

LAST=1
while true
do
    ping -c 1 $WAN_PING
    PING=$?
    if [ $PING != $LAST ]; then
        LAST=$PING
        if [ $PING = 0 ]; then
            /etc/init.d/ppp stop
            sleep 3
            /sbin/route add default gw $WAN_GATEWAY
            echo "nameserver $WAN_DNS" > /etc/resolv.conf
            /usr/sbin/contrack -F
            /etc/scripts/ip-up - - - $ETH2_IPADDR
        else
            /etc/scripts/ip-down - - - $ETH2_IPADDR
            /usr/sbin/contrack -F
            /sbin/route del default gw $WAN_GATEWAY
            /etc/init.d/ppp start
        fi
    fi
    sleep 1
done
```

How It Works

- The script defines IP addresses for ping testing (`WAN_PING`), the primary WAN gateway (`WAN_GATEWAY`), and DNS (`WAN_DNS`). These likely need customization.
- `. /etc/settings.eth` : Sources variables from the Ethernet settings file (e.g., `$ETH2_IPADDR`) used later, assuming this relates to the primary WAN interface (`eth1`). The exact contents and relevance of this file depend on the router configuration.
- **Initial Network Setup**
 - Adds a static route for the `WAN_PING` target via the `WAN_GATEWAY` . This ensures ping tests go through the correct interface when the primary WAN is assumed to be up initially.
 - Adds `iptables` NAT rules for the `eth1` interface. The `PREROUTING` rule uses `-j napt` , which is non-standard (standard Linux typically uses `-j DNAT` targets here if needed). The `POSTROUTING` rule enables standard MASQUERADE (Source NAT) for outgoing traffic on `eth1` , excluding ESP protocol packets (common for VPN passthrough). The effectiveness of `-j napt` depends on its implementation in the router's firmware.
- **Monitoring Loop:**
 - `LAST=1` : Initializes the state variable assuming the connection is initially down (non-zero ping status), ensuring the script potentially triggers a switch on the first successful ping if the WAN is already up at boot.
 - `while true` : Enters an infinite loop for continuous monitoring.
 - `ping -c 1 $WAN_PING` : Sends one ping packet to the target address.
 - `PING=$?` : Captures the exit status of the `ping` command (0 for success, non-zero for failure).
 - `if [$PING != $LAST]` : Checks if the connectivity status has changed since the last check. This edge-triggered approach prevents actions from repeating every second if the state remains stable.
 - * `LAST=$PING` : Updates the stored status for the next comparison.
 - * **If WAN came UP (`PING = 0`)**: Executes commands to switch back to the primary Ethernet WAN.
 - `/etc/init.d/ppp stop` : Stops the Mobile WAN (PPP) connection.
 - `sleep 3` : Pauses briefly to allow the PPP connection to terminate cleanly.
 - `/sbin/route add default gw $WAN_GATEWAY` : Adds the default route via the primary WAN gateway.
 - `echo "nameserver $WAN_DNS" > /etc/resolv.conf` : Sets the DNS server configuration to use the primary WAN's DNS.
 - `/usr/sbin/conntrack -F` : Flushes the connection tracking table to remove potentially stale entries related to the PPP connection.
 - `/etc/scripts/ip-up ...` : Executes the standard system IP-up script, potentially for the Ethernet interface (`eth1`). The parameters passed (`eth1` , `$ETH2_IPADDR` , etc.) are examples and depend on what the actual `ip-up` script expects.
 - * **If WAN went DOWN (`PING != 0`)**: Executes commands to switch to the backup Mobile WAN (PPP) connection.

- `/etc/scripts/ip-down ...` : Executes the standard system IP-down script for the Ethernet interface.
- `/usr/sbin/conntrack -F` : Flushes connection tracking.
- `/sbin/route del default gw $WAN_GATEWAY` : Removes the default route via the primary WAN gateway.
- `/etc/init.d/ppp start` : Starts the Mobile WAN (PPP) connection. The PPP daemon is typically configured to establish its own default route and DNS settings when it connects successfully.
- `sleep 1` : Pauses for 1 second before the next ping check. Adjust this value to balance responsiveness and system load.

3.8 Executing AT Commands on Cellular Module



Advantech routers utilize a cellular (GSM/LTE/5G) module for mobile network connectivity, which is a primary function. While the router's firmware handles most module settings for reliable operation, direct interaction via AT commands is possible using specific methods for monitoring or advanced configuration.

This section describes methods for sending AT commands directly to the router's cellular module.

AT-SMS Protocol

The AT-SMS protocol refers to a private set of AT commands supported by the routers, allowing direct access to the cellular module via SMS. This can be used for tasks like sending SMS messages or querying module status and settings remotely. Configuration related to SMS handling via AT commands can be found in the GUI under *Configuration* → *Services* → *SMS*. You can also refer to the [AT Commands \(AT-SMS\)](#) Application Note for details on the specific command syntax.

The `gsmat` Command (Non-exclusive Access)

For monitoring the cellular module or executing basic AT commands without disrupting the router's primary mobile connection, the `gsmat` command is provided.

Usage

Connect to the router via SSH or Telnet and execute `gsmat` followed by the desired AT command string.

```
# Get module information
gsmat ATI

# List all SMS messages (double quotes escaped)
gsmat AT+CMGL=\\\\"ALL\\\\"

# List all SMS messages (alternative: single quotes)
gsmat 'AT+CMGL="ALL"'
```



Notes on `gsmat`

- This provides "Non-exclusive access", meaning the router's firmware continues managing the mobile connection simultaneously.
- It is primarily intended for monitoring. Modifying critical module settings via `gsmat` is generally not recommended as it might conflict with the firmware's management.
- Special characters within the AT command string, such as double quotes (`"`), dollar signs (`$`), and semicolons (`;`), must be properly escaped with backslashes (`\`) or the entire AT command string should be enclosed in single quotes (`'`) to prevent interpretation by the shell.

Exclusive Access

For advanced diagnostics or configuration requiring full control over the module, "Exclusive access" can be used. This method temporarily stops the router's mobile network management, allowing direct, unimpeded communication with the module.



Exclusive access stops the router's primary mobile connectivity and is intended for expert users familiar with AT commands and potential consequences. Improper use can disrupt network service or require a router reboot. Advantech may not provide support for issues arising from the use of exclusive access.

Steps for Exclusive Access

1. Connect to the router via SSH or Telnet.
2. Stop the Mobile WAN service (PPP daemon):



```
service ppp stop
```

3. Start the TCP/USB proxy (`portd`) to forward a TCP port to the module's serial device. The specific TTY device (`/dev/ttyUSB8` , `/dev/ttyS2` , `/dev/ttyACM8` , etc.) varies depending on the router model and cellular module. You can use `dmesg` to identify the correct port. Port 8000 is used in this example.



```
# Example using common port /dev/ttyUSB8 and TCP port 8000
portd -c /dev/ttyUSB8 -t 8000 &
```

4. Connect to the specified TCP port (e.g., 8000) using a Telnet client or similar tool from your computer or another process on the router.



```
# Example connecting from the router itself
telnet localhost 8000
```

5. Once connected via TCP, you can send AT commands directly to the module and view the responses.
6. **Exclusive Access** (`portd`): When finished, stop the `portd` process (e.g., using `killall portd`) and restart the Mobile WAN service (`service ppp start`) to restore normal router operation.

How It Works Summary

- **Non-exclusive Access** (`gsmat`): Sends a single AT command to the module via a firmware intermediary, allowing basic queries without disrupting the main connection. Requires careful shell escaping.
- **Exclusive Access** (`portd`): Stops the router's mobile connection management, runs a proxy daemon (`portd`) to link the module's serial port directly to a TCP port, allowing raw, interactive AT command communication via a TCP client. Requires manual steps to stop/start services and identify the correct serial port.

3.9 Schedule an Automatic Daily Reboot



This script uses the `cron` scheduling daemon to schedule an automatic router reboot at a specific time each day. See Section [2.7 Scheduling Scripts with cron](#) for more details on `cron`.

This example configures the router to automatically reboot daily at 23:55 (11:55 PM).

Startup Script

Place this script in the Startup Script section of the GUI. It adds the reboot job to the system's crontab when the router boots.



```
#!/bin/sh

echo "55 23 * * * root /sbin/reboot" > /etc/crontab
service cron start
```

How It Works

- `echo "55 23 * * * root /sbin/reboot" > /etc/crontab` : This command overwrites the main system crontab file (`/etc/crontab`).
 - `55 23 * * *` : Defines the schedule: 55th minute, 23rd hour (11:55 PM), every day, month, and day of the week.
 - `root` : Specifies the user account under which the command should run.
 - `/sbin/reboot` : The command to execute.



Overwriting `/etc/crontab` directly is generally discouraged as it removes any other system jobs. A safer approach is to add jobs to user-specific crontabs (e.g., in `/etc/crontabs/root`) or use a drop-in directory like `/etc/cron.d/` if supported by the router's `cron` implementation.

- `service cron start` : Attempts to start or restart the `cron` daemon using the `service` utility. This is necessary for `cron` to recognize the new schedule in the crontab file. The exact command to manage the `cron` service might vary (`crond`, `busybox crond`, etc.).
- Once active, the `cron` daemon will execute `/sbin/reboot` daily at 23:55.

3.10 Voltage Drop SMS Alert



- This script monitors the router's supply voltage and sends an SMS alert if the voltage drops below a specified threshold.
- This functionality is only supported by router models capable of measuring supply voltage (Check GUI: *Status* → *System Information* → *Supply Voltage*).
- It uses the router's proprietary utilities (`status` , `led` , `gsmsms`) and should be placed in the Startup Script to run automatically after boot.
- Adjust the voltage threshold (`Umin`) and recipient phone number (`Num`) variables as needed.

This example demonstrates how to receive an SMS notification when the supply voltage falls below a specific level.

Startup Script

Place this script in the Startup Script section of the GUI.

```
#!/bin/sh

mkdir -p /var/voltaged

cat > /var/voltaged/voltaged << EOF
#!/bin/sh

# Specify these power supply threshold
Umin=16.3
# Specify phone number
Num=+420123456789

old=0
while true
do
    Uget=$( status sys | awk '/^Supply Voltage/ { print \$4 }' )
    if [ "$(echo "\$Uget \$Umin" | awk '{print (\$1 < \$2)}')" -eq 1 ]; then
        if [ \$old -eq 0 ]; then
            led on
            sms \$Num "Low voltage! Voltage is only \$Uget V!"
            old=1
        fi
    else
        led off
        old=0
    fi
    sleep 5
done
EOF

chmod +x /var/voltaged/voltaged
sleep 60 # Let establish MWAN connection
/var/voltaged/voltaged &
```



How It Works

- The Startup Script first creates a directory (`/var/voltaged`) to hold the monitoring daemon script.
- It defines the minimum voltage threshold (`Umin`) and the recipient phone number (`Num`).
- A here-document is used to write the actual monitoring logic into the `/var/voltaged/voltaged` file.
- **Inside the monitoring script (`voltaged`):**
 - An infinite loop (`while true`) runs continuously.
 - `status sys | awk '/^Supply Voltage/ print $4 '` : Retrieves the system status and uses `awk` to extract the voltage value (the 4th field on the relevant line).
 - A basic check ensures a voltage value was actually retrieved.
 - `echo "$current_voltage $UMIN_DAEMON" | awk '{print ($1 < $2)}'` : Performs a floating-point comparison using `awk` to check if the current voltage is less than the threshold. `awk` prints `1` if true, `0` if false.
 - **If Voltage is Low (`is_low` is 1):**
 - * It checks if an alert has already been sent (`alert_sent` is 0).
 - * If no alert was sent, it logs the low voltage event, turns the USR LED on (`led on`), sends an SMS using the `sms` command, and sets `alert_sent` to 1 to prevent repeated SMS for the same low-voltage event.
 - **If Voltage is OK (`is_low` is 0):**
 - * It checks if an alert *was* previously sent (`alert_sent` is 1).
 - * If so, it logs the recovery, turns the USR LED off (`led off`), and resets `alert_sent` to 0, allowing a new alert if the voltage drops again later.
 - `sleep 5` : The script pauses for 5 seconds before the next voltage check.
- The Startup Script makes the daemon script executable (`chmod +x`).
- It includes a `sleep 60` to wait for the system (especially the Mobile WAN connection needed for SMS) to potentially initialize before starting the monitoring.
- Finally, it launches the `voltaged` script in the background (`&`).

Part II.

Python

4. Python on Advantech Routers

This chapter describes how to install and utilize Python on Advantech routers by leveraging Python Router Apps (RAs). Python extends the router's capabilities, enabling custom scripting, automation, data processing, and more.

4.1 Introduction to Python Support

Python is a versatile and powerful high-level programming language known for its readability and extensive libraries. On Advantech routers, Python support is provided through installable Router Apps, transforming your router into a more flexible and programmable device. This allows for:

- Automation of router management tasks.
- Custom data collection and analysis directly on the edge.
- Development of IoT solutions and integrations.
- Enhanced network monitoring and diagnostics.

Two versions of the Python Router App are available to cater to different needs and resource constraints, as detailed in the next section.

4.2 Choosing Your Python Router App

Advantech offers two distinct Python Router Apps. Understanding their differences is key to selecting the appropriate one for your requirements.

4.2.1 Python 3 Router App (Full Version)

The Python 3 Router App delivers a full-featured Python 3 environment, empowering your router with advanced development tools. This version includes:

- **pip** : The Python package installer, allowing you to easily install and manage third-party libraries from the Python Package Index (PyPI).
- **Native hashlib** : Provides faster cryptographic operations as it utilizes underlying C libraries, beneficial for performance-sensitive tasks involving hashing.
- **Full UNICODE support**: Essential for applications dealing with international character sets and diverse text data.
- **venv** : The standard tool for creating lightweight, isolated virtual environments, allowing you to manage dependencies for different projects separately.

This version transforms the router into a powerful platform suitable for complex network automation, data analysis, and sophisticated IoT solutions. It is ideal for developers who need a robust and flexible Python environment and where router resources (CPU, RAM, storage) are sufficient.

4.2.2 Python 3 Lite Router App

The Python 3 Lite Router App provides a lightweight Python 3 runtime, optimized for routers with limited resources or for scenarios where a minimal footprint is critical. Key features:

- **Core Python functionality:** Retains the essential Python 3 features for scripting and automation.
- **No `pip`:** This version does **not** include the `pip` package installer. Third-party libraries must be manually managed if needed, or pure-Python libraries can be bundled with your scripts.
- **Limited UNICODE support:** UNICODE support might be limited compared to the full version, which could be a consideration for applications processing diverse text.
- **Pure-Python `hashlib`:** Uses a pure-Python implementation of `hashlib`. While fully functional, it may be slower for intensive cryptographic operations compared to the native version. This ensures compatibility and reduces dependencies in constrained environments.

The Lite version is best suited for simpler tasks such as log parsing, basic device monitoring, or straightforward automation scripts where minimal system impact and resource usage are paramount.

4.2.3 Comparison Summary

The table below summarizes the key differences between the two Python Router Apps:

Feature	Python 3 Router App	Python 3 Lite Router App
Python Version	Python 3.x	Python 3.x
<code>pip</code>	Included	Not Included
<code>hashlib</code>	Native (faster)	Pure-Python (compatible)
UNICODE Support	Full	Limited
<code>venv</code>	Included	Not Included
Resource Usage	Higher	Lower
Ideal Use Case	Complex tasks, development	Simple tasks, resource-constrained

Table 1.: Comparison of Python Router Apps

4.3 Installing Python Router Apps

Python is installed on Advantech routers by installing the appropriate Python Router App package.

4.3.1 Prerequisites

Before installing a Python Router App, please ensure:

- **Sufficient Storage Space:** Python and its libraries can consume significant storage. Available disk space can be checked in the *Router App* GUI.
- **Internet Connectivity (Optional):** If you plan to use `pip` (with the full Python 3 Router App) to download packages, the router will need internet access. The Router App package itself might be obtained via download or provided by Advantech.

4.3.2 Installation Procedure

To install *Python 3* or *Python 3 Lite*, follow the instructions in the Configuration Manual, section *Customization* → *Router Apps*.

4.3.3 Verifying the Installation

Once installed, you can verify that Python is available and running correctly:

1. Access the router's command-line interface (CLI).
2. Type the command to check the Python version:

```
python3 --version
```

This should output the installed Python version, e.g., `Python 3.x.y`.

3. (Optional, for Full Python 3 RA) Test `pip`:

```
pip3 --version
```

4. (Optional) Try to import a standard module like `hashlib` within Python:

```
python3 -c "import hashlib; print(hashlib)"
```

This should print information about the `hashlib` module.

If these commands execute without error, Python is ready to use.

4.4 Accessing and Running Python

You can interact with Python on the router in two primary ways: through the interactive Python shell or by executing Python scripts.


4.4.1 Interactive Python Shell (REPL)

The Read-Eval-Print Loop (REPL) allows you to type Python code directly and see immediate results. This is useful for testing small snippets of code or exploring Python features.

1. Log in to the router's CLI.
2. Start the Python 3 interpreter: `python3`

You should see the Python prompt (`>>>`).

3. You can now type Python commands:



```
>>> print("Hello from Advantech Router!")
Hello from Advantech Router!
>>> a = 10
>>> b = 20
>>> print(a + b)
30
>>> import os
>>> os.uname()
(system information will be displayed here)
```

4. To exit the Python REPL, type:

```
>>> exit()
```

Or press **Ctrl-D** .

4.4.2 Executing Python Scripts


For more complex tasks, you will write Python code into `.py` files and execute them as scripts.

Creating/Transferring Scripts:

- **Using `vi`** : You can create or edit scripts directly on the router using the `vi` text editor:
`vi myscript.py` , refer to [Busybox vi tutorial](#).
- **Transferring Scripts:** You can write scripts on your development computer and transfer them to the router using `scp` (Secure Copy Protocol) or by mounting a USB drive. Example using `scp` from your computer to the router's `/tmp` directory: `scp myscript.py admin@ROUTER_IP_ADDRESS:/tmp/` .
 Note that the `/tmp` directory is deleted upon router restart.

Script Structure and Execution:

1. **Shebang Line (Recommended):** Start your script with a "shebang" line to specify the interpreter. This allows the script to be executed directly.



```
#!/usr/bin/python3

# Your Python code follows
print("This is my Python script.")
```

2. **Make the script executable:**

```
chmod +x /path/to/your/myscript.py
```

3. **Run the script:** If you used a shebang line and made the script executable:


You can always run it by explicitly calling the Python interpreter:

```
python3 /path/to/your/myscript.py
```

4.4.3 Passing Command-Line Arguments

You can pass arguments to your Python scripts from the command line. These are accessible within Python via the `sys.argv` list. Here's an example script

`myscript_args.py` :




```
#!/usr/bin/python3
import sys

print(f"Script name: {sys.argv[0]}")
if len(sys.argv) > 1:
    print(f"First argument: {sys.argv[1]}")
if len(sys.argv) > 2:
    print(f"Second argument: {sys.argv[2]}")
```

To run this script with arguments, execute:

```
python3 myscript_args.py arg1 "another argument"
```

The output of this script will be:



```
Script name: myscript_args.py
First argument: arg1
Second argument: another argument
```

4.4.4 Introduction to Python Scripting

This section provides a very brief overview of Python concepts particularly relevant for scripting. It is not a comprehensive Python tutorial. For in-depth learning, please refer to the official Python documentation at <https://docs.python.org/3/>.


Core Concepts:

- **Variables and Data Types:**

- Strings: `my_string = "hello"`
- Numbers: Integers (`count = 10`), Floats (`pi = 3.14`)
- Lists: Ordered collections, mutable. `my_list = [1, "two", 3.0]`
- Dictionaries: Key-value pairs, unordered, mutable.
`my_dict = {"name": "router", "ip": "192.168.1.1"}`
- Booleans: `True`, `False`


- **Control Flow:**

- Note the code block definition by **indentation** (usually 4 spaces).
- `if/elif/else` : Conditional execution.




```
if status == "up":
    print("System is operational.")
elif status == "down":
    print("System is down!")
else:
    print("Status unknown.")
```

- `for` loops: Iterate over sequences.




```
for item in my_list:
    print(item)
```

- `while` loops: Repeat as long as a condition is true.



```
count = 0
while count < 5:
    print(count)
    count += 1
```


- **Functions:** Define reusable blocks of code (note the indentation).



```
def greet(name):
    print(f"Hello, {name}!")

greet("Advantech User")
```

- **Importing Modules:** Use Python's extensive standard library or third-party modules.



```
import os
import subprocess
import sys
import re
import datetime
import socket
import json

print(os.name)
print(sys.version)

# For the full Python RA, you might import libraries installed via pip
# import requests
```

Standard Library Modules for Scripting:

Here are examples of some standard library modules preinstalled with Python on the router:

- `os` : Interacting with the operating system (e.g., `os.system()` , `os.environ` , file system operations).
- `subprocess` : Recommended for running external shell commands and managing their input/output, especially in BusyBox environments (e.g., `subprocess.run()` , `subprocess.check_output()`).
- `sys` : Access to system-specific parameters and functions, like command-line arguments (`sys.argv`), exit codes (`sys.exit()`).
- `re` : Regular expressions for powerful text pattern matching and manipulation (e.g., parsing log files or command output).
- `datetime` : For working with dates and times (e.g., timestamping logs).
- `socket` : For low-level network operations if needed.
- `json` : For parsing and generating JSON data, common in APIs and configurations.

Non-standard Library Modules for Scripting:

You can install additional modules using `pip3` , refer to Chapter [4.5.1 Using `pip` to Install Third-Party Libraries](#).

4.5 Advanced Features

The full *Python 3 Router App* includes additional tools that enhance development capabilities. These features are not available in the *Python 3 Lite Router App*.

4.5.1 Using `pip` to Install Third-Party Libraries

`pip` , or `pip3` for Python3, is the standard package installer for Python. It allows you to download and install packages from the Python Package Index (PyPI) and other repositories.

Basic Usage:

- **Install a package:**

```
pip3 install package_name
```

For example, to install the popular `requests` library for making HTTP requests:

```
pip3 install requests
```

- **Install a specific version of a package:**

```
pip3 install package_name==1.2.3
```

- **Upgrade an installed package:**

```
pip3 install --upgrade package_name
```

- **List installed packages:**

```
pip3 list
```


- **Show information about an installed package:**

```
pip3 show package_name
```

- **Uninstall a package:**

```
pip3 uninstall package_name
```

Considerations:

- **Storage Space:** Third-party libraries can consume significant storage space. Be mindful of the router's limited resources. Install only necessary packages.
- **Internet Connectivity:** The router needs internet access to download packages from PyPI.
- **Compilation:** Some Python packages may require compilation of C/C++ extensions during installation. The router environment may lack the necessary compilers or development headers. Prioritize packages that are pure Python or provide pre-compiled "wheels" for ARM Linux (or the router's architecture).
- **Permissions:** You typically need root or administrative privileges to install packages globally.

4.5.2 Using `venv` for Isolated Virtual Environments

`venv` is a module used to create isolated Python virtual environments. Each virtual environment has its own Python binary (or a link to it) and can have its own independent set of installed Python packages in its site directories.

Benefits:

- **Dependency Management:** Avoids conflicts between projects that require different versions of the same library.
- **Clean Global Environment:** Keeps your global Python site-packages directory clean.
- **Reproducibility:** Makes it easier to replicate a project's environment.

Basic Usage:

1. **Create a virtual environment:** Navigate to your project directory (or where you want to create the environment) and run:

```
python3 -m venv my_project_env
```

This will create a directory named `my_project_env` (or your chosen name) containing the virtual environment.

2. **Activate the virtual environment:** Before you can use the virtual environment, you need to activate it. The activation script is located in the environment's `bin` directory.

```
source my_project_env/bin/activate
```

Your shell prompt will usually change to indicate that the virtual environment is active (e.g., `(my_project_env) user@router:~$`).

3. **Install packages within the environment:** Once activated, any `pip3 install` commands will install packages into the active virtual environment, not globally.

```
(my_project_env) $ pip3 install requests
```

4. **Run Python scripts:** Python scripts run while the environment is active will use the environment's Python interpreter and its installed packages.
5. **Deactivate the virtual environment:** When you are finished working in the virtual environment, you can deactivate it:

```
(my_project_env) $ deactivate
```

Your shell prompt will return to normal.

6. **Delete the virtual environment:** To completely delete the whole environment, just delete its directory:

```
rm -rf my_project_env
```

Considerations on Routers:

- **Storage:** Each virtual environment duplicates some files or creates symlinks, and stores its own packages, consuming additional storage space. Use judiciously on resource-constrained routers.
- **Activation:** Remember to activate the correct environment before running scripts that depend on its specific packages. For automated scripts (e.g., cron jobs), you'll need to source the activate script or call the Python interpreter from within the venv directly (e.g., `/path/to/my_project_env/bin/python your_script.py`).

4.6 Router-Specific Python Development Notes

Developing Python scripts for Advantech routers requires consideration of their specific operating environment, which is typically based on BusyBox and has resource constraints. Here are key points to keep in mind:

- **Limited Shell Commands and BusyBox Environment:** BusyBox provides a compact set of Unix utilities that often have fewer options and may exhibit slightly different behavior compared to their full GNU counterparts found on standard Linux distributions. Python scripts calling external commands should account for these differences.
- **Robust Command Execution with `subprocess`:** Python's `subprocess` module is highly recommended for running external shell commands. It offers superior control over input/output streams, error handling, and overall execution flow compared to older methods like `os.system()`. Using `subprocess` can also enhance script portability when dealing with variations in command behavior or availability.
- **On-Router Text Editing:** The primary text editor available directly on the router is usually `vi` (or a similar minimalist editor like `nano` on some builds). Familiarity with `vi` is beneficial for quick on-router script modifications. A helpful [BusyBox vi tutorial](#) can be found online.
- **File System Paths and Volatility:** Be mindful of the router's file system structure. Temporary files should typically be written to `/tmp`, which is often a RAM disk (`tmpfs`) and its contents are cleared on reboot. Locations for persistent storage depend on the router model and configuration (e.g., `/opt`, `/mnt/user`, or an attached USB drive).
- **Permissions and Privileges:** Standard Linux file permissions apply. Your Python scripts will need execute permissions (`chmod +x script.py`) to be run directly. Accessing certain system files, network interfaces, or performing privileged operations (like using raw sockets or modifying system configuration) may require root privileges. Run scripts with the minimum necessary privileges.

- **Resource Constraints (CPU, RAM, Storage):** Embedded routers inherently have limited CPU power, RAM, and persistent storage compared to desktop systems. Write efficient Python code. Avoid memory-intensive operations, very large libraries, or frequent disk writes if possible, especially when using the "Lite" version of the Python Router App, which has a smaller footprint. Monitor resource usage during development and testing.
- **Router-Specific SDK Python Module (`um`):** For interacting with router-specific hardware and software functionalities—such as accessing GPIOs, reading system parameters (e.g., cellular signal strength, device temperature), managing expansion port configurations, or generating HTML content for the router's web interface—Advantech provides a specialized Python module named `um`. This module is included as part of the Advantech SDK. Always consult the SDK documentation for your target platform (refer to Section [8.3 SDK \(Software Development Kit\)](#)) for details on the availability, specific features, and usage of the `um` module.
- **Cross-Compilation for Python Packages with C Extensions:** If your Python project requires packages that include C extensions (which are common for performance-critical libraries or those interfacing with C libraries), `pip` running on the router might be unable to build them directly if a C compiler and the necessary development headers are not available on the router (which is typical). In such cases, you may need to:
 - **Cross-compile** the package on a development machine using a toolchain that targets the router's architecture (e.g., ARM). This involves setting up a cross-compilation environment for Python extensions.
 - **Find pre-compiled wheels** (`.whl` files) for the package that are specifically built for the router's architecture and Python version.
 - Consider if a pure-Python alternative to the package exists.

This is an advanced topic and generally falls outside simple scripting.


By leveraging Python's strengths in scripting and its `subprocess` module, you can effectively overcome many of the limitations of a basic BusyBox shell and create powerful automation and management tools for your Advantech router.

5. Practical Python Script Examples

The following examples demonstrate simple use cases for Python on your Advantech router. These scripts assume they are run from the router's CLI.

5.1 Gathering System Information

This script uses the `subprocess` module to run BusyBox commands and display system information. This script pings a list of IP addresses to check their reachability. Save the code below into a file named `system_info.py`.



```
#!/usr/bin/python3
import subprocess

print("Gathering Basic System Information...\n")

print("--- System Uptime ---")
try:
    with open("/proc/uptime", "r") as f:
        uptime_seconds = float(f.readline().split()[0])

    days = int(uptime_seconds // (24 * 3600))
    uptime_seconds %= (24 * 3600)
    hours = int(uptime_seconds // 3600)
    uptime_seconds %= 3600
    minutes = int(uptime_seconds // 60)
    seconds = int(uptime_seconds % 60)

    print(f"System has been up for: {days} days, {hours} hours, {minutes} minutes, {seconds} seconds.")
except Exception as e:
    print(f"Could not get uptime: {e}")
print()

print("--- Memory Usage ---")
try:
    result = subprocess.run("free", shell=True, capture_output=True, text=True, check=True)
    print(result.stdout.strip())
    print("(Output is typically in kilobytes)")
except Exception as e:
    print(f"Could not get memory usage: {e}")
print()

print("--- Disk Space ---")
try:
    result = subprocess.run("df -k", shell=True, capture_output=True, text=True, check=True)
    print(result.stdout.strip())
    print("(Output is in 1K-blocks/kilobytes)")
except Exception as e:
    print(f"Could not get disk space: {e}")
print()

print("--- Hostname ---")
try:
    with open("/proc/sys/kernel/hostname", "r") as f:
        hostname = f.read().strip()
    print(hostname)
except Exception as e:
    print(f"Could not get hostname: {e}")
print()
```

How It Works

This script gathers and displays basic system information from an Advantech router.

- The script begins with a shebang `#!/usr/bin/python3`, indicating it should be executed using the Python 3 interpreter located at `/usr/bin/python3`.
- It imports the `subprocess` module: This standard Python module is used to run external shell commands, such as `free` and `df -k`, and capture their output.
- The script then proceeds sequentially to gather and print different pieces of system information, each within its own section. For error handling, each section uses a `try...except Exception as e:` block, which catches any general error that occurs and prints a user-friendly message.

- **System Uptime:**

- A header `"--- System Uptime ---"` is printed.
- The script reads uptime information directly from the `/proc/uptime` file. This is a standard virtual file in Linux systems that provides uptime statistics without needing an external `uptime` command.
 - * `with open("/proc/uptime", "r") as f:` : This opens the file in read mode (`"r"`). The `with` statement ensures the file is automatically closed even if errors occur.
 - * `f.readline().split()[0]` : Reads the first line from the file, splits this line into a list of strings (words), and takes the first element (`[0]`), which is the total system uptime in seconds.
 - * `float(...)` : Converts the uptime string to a floating-point number.
- The total uptime in seconds is then converted into a more human-readable format of days, hours, minutes, and seconds using integer division (`//`) and modulo (`%`) operations.
- The formatted uptime string is printed to the console.

- **Memory Usage:**

- A header `"--- Memory Usage ---"` is printed.
- The `subprocess.run()` function is used to execute the external shell command `free`.
 - * Command: `"free"` (displays amount of free and used memory in the system).
 - * `shell=True` : Indicates that the command should be executed through the system's shell.
 - * `capture_output=True` : Specifies that the standard output and standard error of the command should be captured.
 - * `text=True` : Decodes the captured output as a text string.
 - * `check=True` : If the command returns a non-zero exit status (indicating an error), a `CalledProcessError` exception is raised.
- `result.stdout.strip()` : The captured standard output from the `free` command is printed after removing any leading or trailing whitespace.
- A note `"(Output is typically in kilobytes)"` is appended, as this is a common default for the BusyBox version of `free`.

- **Disk Space:**

- A header `"--- Disk Space ---"` is printed.
- Again, `subprocess.run()` is used, this time to execute the command `df -k`.
 - * Command: `"df -k"` (reports file system disk space usage). The `-k` option ensures the output is in 1K-blocks (kilobytes), which is generally supported by BusyBox `df` and avoids issues with unsupported options like `-h`.
- The standard output from `df -k` is printed.
- A note `"(Output is in 1K-blocks/kilobytes)"` clarifies the units of the displayed sizes.

- **Hostname:**

- A header `"--- Hostname ---"` is printed.
 - The system's hostname is read directly from the `/proc/sys/kernel/hostname` file.
 - * `with open("/proc/sys/kernel/hostname", "r") as f:` : Opens this virtual file for reading.
 - * `f.read().strip()` : Reads the entire content of the file (which is the hostname) and removes any leading/trailing whitespace.
 - The retrieved hostname is printed to the console.
- After each section's output (or error message), `print()` is called to produce a blank line, improving the readability of the overall output in the terminal.

Script Testing

Below is the console output, displaying the script's creation (using `vi`) and its testing.

```
/var/scripts # vi system_info.py
(...create and save scripty with pi...)
/var/scripts # chmod +x system_info.py
/var/scripts # ./system_info.py
Gathering Basic System Information...

--- System Uptime ---
System has been up for: 0 days, 1 hours, 10 minutes, 34 seconds.

--- Memory Usage ---
total      used      free      shared buff/cache   available
Mem:      503840    38604    448744      184    16492    456496
Swap:      0         0         0
(Output is typically in kilobytes)

--- Disk Space ---
Filesystem      1024-blocks    Used Available Use% Mounted on
/dev/root        64512    23544    40968   36% /
devtmpfs        251408         0    251408    0% /dev
none            251920         0    251920    0% /tmp
none            50384         0    50200    0% /var
/dev/mtdblock7  131072    19812    111260   15% /opt
/dev/mtdblock8   128         36         92   28% /var/data
(Output is in 1K-blocks/kilobytes)

--- Hostname ---
Router

/var/scripts #
```

5.2 Basic Network Reachability Test

This script pings a list of IP addresses to check their reachability. Save the code below into a file named `network_test.py`.



```
#!/usr/bin/python3
import subprocess
import sys

def ping_host(host_ip, count=1):
    """Pings a host and returns True if reachable, False otherwise."""
    command = ["ping", "-c", str(count), "-W", "1", host_ip]
    try:
        process = subprocess.Popen(command, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)
        process.communicate()
        return process.returncode == 0
    except FileNotFoundError:
        print(f"Error: 'ping' command not found.", file=sys.stderr)
        return False
    except Exception as e:
        print(f"Error pingping {host_ip}: {e}", file=sys.stderr)
        return False

if __name__ == "__main__":
    hosts_to_check = ["8.8.8.8", "1.1.1.1", "192.168.1.254"]

    if len(sys.argv) > 1:
        hosts_to_check = sys.argv[1:]

    print("--- Network Reachability Test ---")
    for host in hosts_to_check:
        print(f"Pinging {host}... ", end="")
        if ping_host(host):
            print("Reachable")
        else:
            print("Unreachable")
```

How It Works

This script checks the network reachability of one or more specified hosts by sending ICMP ECHO_REQUEST packets (commonly known as "pinging" them).


- The script starts with a shebang `#!/usr/bin/env python3`, which tells the system to use the `python3` interpreter found in the user's environment PATH to execute the script.
- It imports two standard Python modules:
 - `subprocess`: Used to create and manage child processes, specifically to run the external `ping` command.
 - `sys`: Provides access to system-specific parameters and functions, such as command-line arguments (`sys.argv`) and standard error stream (`sys.stderr`).
- **The `ping_host(host_ip, count=1)` Function:** This function is responsible for pinging a single host and determining if it is reachable.
 - It takes two arguments:
 - * `host_ip`: The IP address or hostname of the target to ping.

- * `count=1` : An optional argument specifying the number of ping packets to send. It defaults to 1.
- A list named `command` is constructed, representing the `ping` command and its arguments:
 - * `["ping", "-c", str(count), "-W", "1", host_ip]`
 - * `"ping"` : The ping utility.
 - * `"-c", str(count)` : Sends a specific `count` of packets. (e.g., `-c 1` sends one packet).
 - * `"-W", "1"` : Sets a timeout of 1 second to wait for each reply. This is crucial for BusyBox ping, which might behave differently from other ping versions regarding timeout for the whole operation vs. per-packet.
 - * `host_ip` : The target host.
- The `try...except` block handles potential errors during the ping process:
 - * `subprocess.Popen(command, stdout=subprocess.DEVNULL, stderr=subprocess.DEVNULL)` : This starts the `ping` command as a new process.
 - `stdout=subprocess.DEVNULL` and `stderr=subprocess.DEVNULL` : The standard output and standard error streams of the `ping` command are redirected to `DEVNULL`, meaning its output will not be displayed on the console. The script only cares about the success or failure (exit code).
 - * `process.communicate()` : Waits for the `ping` command to complete.
 - * `return process.returncode == 0` : Checks the exit code of the `ping` command. An exit code of `0` typically indicates success (host is reachable). The function returns `True` if successful, `False` otherwise.
 - * `except FileNotFoundError:` : If the `ping` command itself is not found on the system, this error is caught. An error message is printed to `sys.stderr`, and the function returns `False`.
 - * `except Exception as e:` : Catches any other exceptions that might occur during the ping attempt. An error message including the specific exception is printed to `sys.stderr`, and the function returns `False`.
- **Main Execution Block** (`if __name__ == "__main__":`): This part of the script runs when the script is executed directly.
 - `hosts_to_check = ["8.8.8.8", "1.1.1.1", "192.168.1.254"]` : A default list of IP addresses/hostnames is defined. These are public DNS servers and a common local gateway IP.
 - Command-Line Argument Handling:
 - * `if len(sys.argv) > 1:` : Checks if the script was run with any command-line arguments (the script name itself is `sys.argv[0]`).
 - * `hosts_to_check = sys.argv[1:]` : If arguments are provided, the default list is replaced with the list of arguments supplied by the user. For example, running `python3 script.py google.com example.com` would set `hosts_to_check` to `["google.com", "example.com"]`.

- A header `"--- Network Reachability Test --- "` is printed.
- The script then iterates through each `host` in the `hosts_to_check` list:
 - * `print(f"Pinging host... ", end="")` : Prints a message indicating which host is currently being pinged. The `end=""` argument prevents a newline, so the "Reachable" or "Unreachable" status will appear on the same line.
 - * `if ping_host(host):` : Calls the `ping_host` function with the current host.
 - * Based on the boolean return value of `ping_host()` , it prints either `"Reachable"` or `"Unreachable"` to the console.

Script Testing

Below is the console output, displaying the script's creation (using `vi`) and its testing.



```
/var/scripts # vi network_test.py
(...create and save script with pi...)
/var/scripts # chmod +x network_test.py
/var/scripts # ./network_test.py
--- Network Reachability Test ---
Pinging 8.8.8.8... Reachable
Pinging 1.1.1.1... Reachable
Pinging 192.168.1.254... Unreachable
/var/scripts #
```

5.3 Simple Log File Monitoring

This script demonstrates reading a log file (e.g., `/var/log/messages`) and searching for specific keywords. Save the code below into a file named `log_monitoring.py`.

Continuously reading large log files can impact performance. This example is illustrative. For production use, consider more optimized log monitoring tools or techniques if available or develop a more efficient file tailing mechanism in Python.

```
#!/usr/bin/python3
import sys
import re
import time

def monitor_log(log_file_path, keywords, interval_seconds=5):
    """Monitors a log file for lines containing specified keywords."""
    print(f"Monitoring {log_file_path} for keywords: {keywords}")
    print(f"Checking every {interval_seconds} seconds. Press Ctrl+C to stop.")

    patterns = [re.compile(keyword, re.IGNORECASE) for keyword in keywords]

    try:
        last_lines_seen = set()
        while True:
            current_lines = set()
            try:
                with open(log_file_path, 'r') as f:
                    for line_number, line in enumerate(f, 1):
                        line = line.strip()
                        current_lines.add(line)
                        if line in last_lines_seen:
                            continue

                        for pattern in patterns:
                            if pattern.search(line):
                                print(f"[MATCH] Line {line_number}: {line}")
                                break
            except FileNotFoundError:
                print(f"Error: Log file '{log_file_path}' not found.", file=sys.stderr)
                return
            except Exception as e:
                print(f"Error reading log file: {e}", file=sys.stderr)

            last_lines_seen.update(current_lines)
            time.sleep(interval_seconds)

    except KeyboardInterrupt:
        print("\nLog monitoring stopped.")

if __name__ == "__main__":
    log_file = "/var/log/messages"
    search_keywords = ["error", "warn", "dhcp"]

    if len(sys.argv) > 1:
        log_file = sys.argv[1]
    if len(sys.argv) > 2:
        search_keywords = sys.argv[2:]

    try:
        with open(log_file, 'r') as f:
            pass
    except Exception as e:
        print(f"Cannot access log file {log_file}: {e}", file=sys.stderr)
        sys.exit(1)

    monitor_log(log_file, search_keywords)
```

How It Works

This script continuously checks a log file for lines matching specified keywords and prints any matches.

- **Setup:**

- It uses the `python3` interpreter, as indicated by `#!/usr/bin/python3`.
- Modules imported: `sys` (for system interaction like command-line arguments), `re` (for regular expression-based keyword matching), and `time` (for delays).

- **Core Function:** `monitor_log(log_file_path, keywords, interval_seconds=5)`

- **Initialization:** Prints what it's monitoring and how often. Keywords are compiled into case-insensitive regular expression patterns (`re.compile(keyword, re.IGNORECASE)`) for efficient searching. A set `last_lines_seen` is used to track already processed lines to avoid re-printing old matches.
- **Monitoring Loop** (`while True:`): This loop runs indefinitely until stopped (e.g., by Ctrl+C).
 - * In each iteration, it attempts to open and read the specified `log_file_path`. A set `current_lines` stores all unique lines from this read.
 - * It iterates through each line of the log file. If a line hasn't been seen before (`not in last_lines_seen`), it checks if any of the compiled keyword patterns match anywhere in the line using `pattern.search(line)`.
 - * If a match is found, the line number and the line content are printed.
 - * After checking all lines, `last_lines_seen` is updated with `current_lines`.
 - * The script then pauses for `interval_seconds` using `time.sleep()` before re-reading the file.
- **Error Handling:**
 - * If the log file is not found during a check, an error is printed, and monitoring for that file stops.
 - * Other file reading errors are also caught and reported.
 - * Pressing Ctrl+C (`KeyboardInterrupt`) stops the monitoring loop gracefully and prints a message.

- **Main Execution Block** (`if __name__ == "__main__":`)

- **Defaults & Arguments:** Sets default `log_file` (e.g., `"/var/log/messages"`) and `search_keywords` (e.g., `["error", "warn"]`). These can be overridden by providing command-line arguments: the first argument for the log file path, and subsequent arguments for keywords.
- **Initial File Check:** Before starting, it tries to open the target log file to ensure it's accessible. If not, it prints an error and exits (`sys.exit(1)`).
- **Start Monitoring:** Calls the `monitor_log` function with the determined log file and keywords.

Script Testing

Below is the console output, displaying the script's creation (using `vi`) and its testing.



```
/var/scripts # vi log_monitoring.py
(...create and save scripty with pi...)
/var/scripts # chmod +x log_monitoring.py
/var/scripts # ./log_monitoring.py
Monitoring /var/log/messages for keywords: ['error', 'warn', 'dhcp']
Checking every 5 seconds. Press Ctrl+C to stop.
[MATCH] Line 15: 2025-05-22 12:34:43 [info] dhcpd: Wrote 0 leases to leases file.
[MATCH] Line 21: 2025-05-22 12:34:48 [warning] tottd[1431]: Disabling rescanning of network interfaces
^C
Log monitoring stopped.
```

Part III.

Router Apps

6. Getting Started with Router Apps

6.1 What are Router Apps

Router App (formerly known as *User Module*) refers to a software application specifically designed to run on Advantech routers. These applications allow users to extend the router's built-in functionality, customize its behavior, and add new features tailored to specific needs. This guide describes the structure, development process, and technical considerations necessary for creating your own Router Apps that integrate correctly with the Advantech router environment.

Advantech routers run a Linux-based operating system (ICR-OS). While using a Linux environment for Router App development is recommended for ease of use with toolchains and testing, it is not strictly required. Router Apps can be developed using languages such as C, C++, or Python, provided they can be compiled or executed within the router's environment. This guide focuses on the general structure, scripting conventions, configuration management, and system integration rules applicable across different development approaches.

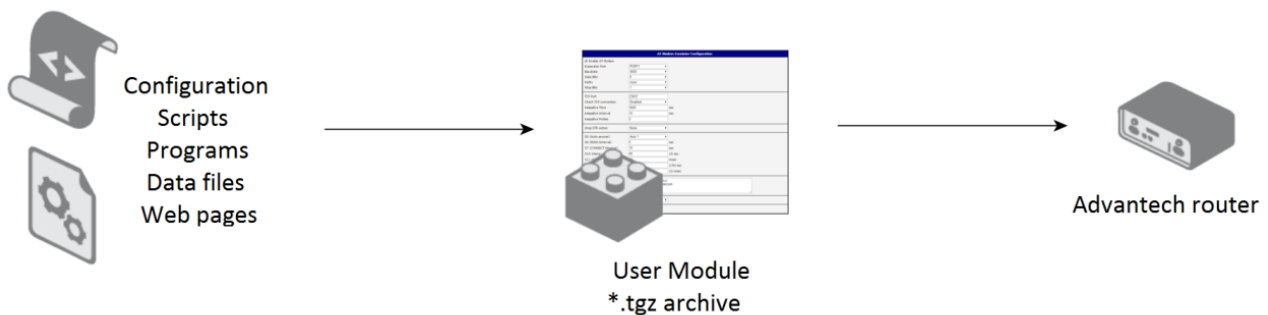


Figure 1.: Router Apps Programming Scheme

6.2 Overview of Development Approaches

Router Apps can broadly be categorized into:

- **Compiled Applications:** Typically written in C or C++, these applications are cross-compiled on a development machine to produce binaries that run directly on the router's processor. They offer maximum performance and low-level system access.
- **Scripted Applications:** Often written in Python or shell scripts, these applications are interpreted at runtime on the router. They offer faster development cycles and ease of use for many tasks.

Refer to Chapter [8.3 SDK \(Software Development Kit\)](#) for information on available Software Development Kits (SDKs) and cross-compilers for C/C++ and section [II](#) for Python development.

6.3 General Development Workflow and Tools

The general workflow for developing a Router App involves:

1. Setting up the development environment (SDK, toolchains).
2. Writing the application code (C/C++, Python, scripts).
3. Creating necessary control scripts (`init` , `install` , etc.) and configuration files.
4. Packaging the application into a `*.tgz` archive.
5. Uploading and testing the application on the target router.
6. Debugging and iterating.

Recommended General Tools

- Basic familiarity with the Linux command line and shell scripting is beneficial for creating installation and initialization scripts.
- A text editor suitable for programming and script creation (ensuring Unix-style line endings, e.g., LF).
- Tools for creating TAR archives and Gzip compression (standard on most Linux distributions, readily available for other operating systems).

Available Commands

Router Apps can leverage many standard Linux commands and utilities provided by the router's BusyBox-based environment. To explore available commands, connect to the router's console (via SSH or Telnet) and press the TAB key twice for shell completion suggestions, or run `busybox --list` . For help on a specific BusyBox command, you can often use `<command> --help` . For details on both standard and Advantech-specific commands, refer to the [Command Line Interface](#) Application Note.

Login Events in the System Log

To aid in debugging your scripts, you can add logging statements that write messages to the router's System Log (viewable in the web interface under *Status* → *System Log*). Use the `logger` utility for this purpose. Add a line like the following near the beginning of each script (e.g., `install` , `uninstall` , `init` , etc.):

```
/usr/bin/logger -t mymodule "DEBUG: $0 $@"
```

Here:

- `-t` mymodule sets the tag name for the log entry (typically the name of your module or application).
- `$0` expands to the script's name.
- `$@` expands to all arguments passed to the script.

This will log a message such as:

```
DEBUG: /etc/rc.d/init.d/myscript start verbose
```

7. Router App Structure

7.1 Directory Structures for Applications

Router Apps, once installed, primarily reside in the `/opt/<RAname>` directory on the router, where `<RAname>` is the name of your Router App (RA).

Recommended location for persistent runtime data generated by your app is `/var/data/<RAname>` directory. This directory is usually cleaned up upon app uninstallation.

Internal Archive Structure

The schema below illustrates the typical internal structure of the RA directory structure.

<RAname>	The base directory for installed RA.
— /etc/	Subdirectory for scripts, information, and configuration files.
— defaults	File containing default configuration entries for the RA.
— depends	File listing other RAs this app depends on.
— description	A more detailed description of the RA, written in several sentences.
— init	Initialization script (handles start, stop, etc.).
— install	Script executed during the installation process.
— ip-up	Script executed when a WAN connection (IPv4) is established.
— ip6-up	Script executed when a WAN connection (IPv6) is established.
— ip-down	Script executed when a WAN connection (IPv4) is lost.
— ip6-down	Script executed when a WAN connection (IPv6) is lost.
— name	File containing the human-readable name for the RA, shown in the web interface.
— report	Script executed during the creation of the report file.
— requires	File specifying the minimum compatible router firmware version.
— settings	Actual configuration file (not included in the <code>*.tgz</code> archive).
— summary	A brief one-sentence summary of the RA.
— uninstall	Script executed during the uninstallation process.
— version	File containing the RA version, shown in the web interface.
— /bin/	Subdirectory for auxiliary files, binaries, daemons, or <code>*.cgi</code> scripts.
— /lib/	For private shared libraries used only by your app (if not statically linked).
— /www/	Subdirectory containing web interface files (HTML, CGI, etc.).

File type legend:

Information files Configuration files Script files

All subdirectories and files within the top-level `/opt/<RAname>` directory are optional, except for those necessary for the app's functionality and integration, e.g. `init` script if the app needs to start/stop/restart.

Information Files

These **optional plain text** files provide metadata about the Router App, used by the router's management interface.

depends

This file lists dependencies, specifying any other Router Apps required for this app to function correctly. List one Router App name per line. The name must match the directory name (`<RAname>`) of the dependency as it appears in its `*.tgz` archive.



```
Python
otherModuleName
```

description

This file contains a more detailed description of the Router App, written in several sentences. This description is not visible in the router GUI.

name

This file contains the full, human-readable name of the Router App. This name will be displayed in the router's web interface. It is recommended to use only the characters 'a'-'z', 'A'-'Z', '0'-'9', and space (' ') for the name. If this file is absent, the RA's directory name (`<RAname>`) will be used instead.



```
My Custom Router App
```

requires

This file specifies the minimum required version of the router's firmware compatible with this Router App. The version must follow the three-number format: `MAJOR.MINOR.PATCH` .



```
6.5.0
```

summary

This file contains a brief one-sentence summary of the Router App. This description is not visible in the router GUI.

version

This file contains the version information for the Router App, which will be displayed in the web interface. The recommended format is semantic versioning (`MAJOR.MINOR.PATCH`) followed by a date in `YYYY-MM-DD` format, as shown below. If this file is missing, the version of the Router App will not be displayed in the router's web interface.



```
1.0.0 (2015-07-15)
```

Configuration Files


defaults

This file must contain the default configuration parameters for the Router App. These parameters are used during the initial installation and when the router is reset to factory defaults (using the RST button).

The content of this file should be copied to the `settings` file by the `init` script during installation (specifically, when called with the `defaults` argument) to enable configuration backup functionality. If your Router App does not require configuration, this file is not needed.

Variables must be defined using the following format: `MOD_<RName>_<variable_name>=<value>`

Here, `MOD` signifies Router App, making it distinguishable from the router's core configuration parameters. `<RName>` is the name of the Router App (matching the directory and archive name), and `<variable_name>` is the desired parameter name. It is recommended to use uppercase letters for both `<RName>` and `variable_name`.



```
MOD_MYMODULE_ENABLED=1
MOD_MYMODULE_PARAM1=0
MOD_MYMODULE_PARAM2=5
MOD_MYMODULE_PARAM3=20
```


settings

This file **should not be included** in the Router App's `*.tgz` archive. It should be created during the installation process by the `init` script (when called with the `defaults` parameter), see next paragraph. Typically, the `init` script copies the contents of the `defaults` file into the runtime `settings` file, enabling configuration backup.

When the router's configuration is backed up, the contents of the `settings` file are included in the resulting `*.cfg` file. This file also persists across Router App updates. When updating a Router App, the system backs up the existing `settings` file. The updated Router App will first attempt to use the restored `settings` file. It only refers back to the `defaults` file for parameters not found in the restored `settings` file (e.g., newly added parameters).

Control Scripts

All control scripts (such as `init`, `install`, `uninstall`, `ip-up`, etc., located in the `etc/` directory of your Router App) must start with the shebang line `#!/bin/sh`. They should be written using POSIX-compliant shell syntax to ensure compatibility with the BusyBox ash shell environment typically found on Advantech routers.



Do not forget to set **execute permissions** for all your script files. This should be done in your development environment before packaging the Router App, or within the `install` script itself if appropriate for dynamically generated scripts. Use the command: `chmod +x <script_filename>`.

init

This is the initialization script for the Router App. It is invoked by the system with different parameters depending on the context (e.g., router startup, Router App installation, update, removal). It can also be called manually with a specific parameter. If an `init` script is not present, no actions will be performed for the Router App during these events. The script accepts the following parameters:

- **start** – Executed automatically at router startup and after the Router App is successfully installed. Use this to start daemons or perform necessary setup.
- **stop** – Executed automatically before updating or uninstalling the Router App. Use this to gracefully stop daemons and perform cleanup.
- **restart** – **Not called automatically.** Can be called manually to stop and then start the Router App.
- **status** – **Not called automatically.** Can be called manually to check if the Router App's services are running. Should exit with 0 if running, non-zero otherwise.
- **defaults** – Executed automatically after installation and when the router's RST button is pressed (factory reset). Its primary purpose is to copy the contents of the 'defaults' file into the runtime `settings` file, usually located at `/opt/$MODNAME/etc/settings`.

An example `init` script is shown below. This example simply prints messages into Syslog indicating the action being performed. Set the `MOD_NAME` variable in this script to match your Router App Name. Note the `cp` command in the `defaults` case, which copies default settings to the runtime settings file, enabling configuration backup. This source code can be found in the `example1` of the SDK documentation (reref to Chapter [8.3 SDK \(Software Development Kit\)](#)).

```
#!/bin/sh

MOD_NAME=example1
MOD_DEFAULTS=/opt/$MOD_NAME/etc/defaults
MOD_SETTINGS=/opt/$MOD_NAME/etc/settings
[ -L "$MOD_SETTINGS" ] && MOD_SETTINGS=`readlink $MOD_SETTINGS`

/usr/bin/logger -t $MOD_NAME "DEBUG: $0 $@"

case "$1" in
  start)
    echo "Starting module $MOD_NAME: done"
    exit 0
    ;;
  stop)
    echo "Stopping module $MOD_NAME: done"
    exit 0
    ;;
  restart)
    $0 stop
    $0 start
    ;;
  status)
    echo "Module $MOD_NAME is running"
    exit 0
    ;;
  defaults)
    cp $MOD_DEFAULTS $MOD_SETTINGS 2>/dev/null
    ;;
  *)
    echo "Usage: $0 {start|stop|restart|status|defaults}"
    exit 1
esac
```

install

This script is executed once, immediately after the Router App's files have been extracted and copied to the `/opt/` directory during the installation process. Use this for initial setup tasks that only need to run once upon installation. See Section 7.3 for the sequence of script execution.

uninstall

This script is executed during the uninstallation process. It runs after the Router App has been stopped (via `init stop`) but just before its files are deleted from the `/opt/` directory. Use this script to perform any necessary cleanup. See Section 7.3 for the script execution order.

ip-up

Executed when a WAN (IPv4) connection is established.

Parameters: `<ip-address> <WAN-interface>` .

Example: `/opt/mymodule/etc/ip-up 10.40.28.64 ppp0`

ip6-up

Executed when a WAN (IPv6) connection is established.

Parameters: `<ip6-address> <WAN-interface>` .

Example: `/opt/mymodule/etc/ip6-up fc00::a40:37 ppp0`

ip-down

Executed when a WAN (IPv4) connection is lost.

Parameters: `<ip-address> <WAN-interface>` .

Example: `/opt/mymodule/etc/ip-down 10.40.28.64 ppp0`

ip6-down

Executed when a WAN (IPv6) connection is lost.

Parameters: `<ip6-address> <WAN-interface>` .

Example: `/opt/mymodule/etc/ip6-down fc00::a40:37 ppp0`

report

This script is executed during the creation of the report file (from console or GUI). It has no parameters passed to the script.

Web Interface Files

Overview of Web Interface Integration

The `/opt/<RAppName>/www/` directory in your Router App package is key for web interface integration. If a file named `index.html` or `index.cgi` (or similar standard index file) exists within this directory, a link to it will appear in the router's main web interface under the *Customization -> Router Apps* section. Clicking this link will access the Router App's web interface.

If the `/www` directory is absent, or if it does not contain a recognizable index file, no link will be shown in the main web interface. The contents of the `/opt/<RAppName>/www` directory are mapped to the following URL: `http(s)://<router ip address>/module/<RAppName>` (where `<RAppName>` is your Router App's name).


Adding Static and Dynamic Content

- **Static Content:** HTML, CSS, JavaScript files, and images can be placed directly into the `/www` directory.
- **Dynamic Content:** CGI scripts (written in shell, Python, C/C++ compiled to executable, etc.) can be placed in `/www`. Ensure they are executable and produce valid HTTP headers and HTML content. The Python example in Section 8.6 is one such case.

Web Interface Security

Regarding access control for the Router App's web interface, you have two options:

1. **Secured (Recommended):** Protect the web interface using the router's existing user authentication system. To do this, create a file named `.htpasswd` inside the `/opt/<RAppName>/www/` directory. This file should be a symbolic link to the router's main password file, located at `/etc/htpasswd`. Use the following command within your `install` script or manually via SSH (execute inside `/opt/<RAppName>/www/`):



```
ln -s /etc/htpasswd .htpasswd
```

This ensures that accessing the Router App's web interface requires the same username and password used to log into the main router interface.

2. **Unsecured:** If no `.htpasswd` file (or a symbolic link named `.htpasswd`) exists in the `/www` directory, the web interface will be accessible to anyone who can reach the router's IP address, without requiring authentication. **This option is strongly discouraged** due to security risks.

7.2 Application Packaging

Router App Archive Format

To upload a Router App into an Advantech router, it must be packaged as a `*.tgz` archive file (a TAR archive compressed with Gzip). This archive **must** contain a single top-level directory, refer to [7.1 Internal Archive Structure](#).

The name of this single top-level directory inside the archive must be identical to the base name of the `*.tgz` archive file itself (excluding the platform suffix and `.tgz` extension). This base name is restricted to a maximum of 24 characters and can only contain alphanumeric characters ('a'-'z', 'A'-'Z', '0'-'9') and the underscore ('_'). Using spaces or other special characters in the base name, or in any subdirectory or file names within the archive, is strongly discouraged.

Archiv name of a Router App has this syntax: `<RAName>.<platform>.tgz` , e.g. `mymodule.v4.tgz` .

Creating a Router App Archive

It is recommended to use our SDK to compile the source code and build the Router App archive. Refer to section [8.4](#).

If you want to create the Router App archive manually, you can proceed as follows. Assume that your Router App files are organized within a directory named `mymodule` and, for demonstration purposes, have a minimal structure containing only the `init` file with the content from section [7.1](#). The source file structure would then look like this:

```
mymodule/          (RA root directory)
`-- etc/           (/etc folder)
    |-- init       (file with init script)
```

To create the archive, you can use the script bellow (named as `pack.sh`).

```
#!/bin/bash
set -e

[ -z "$1" ] && { echo "Error: App name is missing."; echo "Usage: $0 <appname> <platform>"; exit 1; }
[ -z "$2" ] && { echo "Error: Platform is missing."; echo "Usage: $0 <appname> <platform>"; exit 1; }
[ -d "$1" ] || { echo "Error: Directory '$1' not found."; exit 1; }

echo "Packing $1 for platform $2 → $1.$2.tgz"
tar -c --owner=0 --group=0 --mtime="2001-01-01 UTC" --exclude-vcs "$1" | gzip -n > "$1.$2.tgz"
echo "Done."
```

This script has following syntax: `pack.sh <appname> <platform>` . To pack `mymodule` Router App for platform `v4` , navigate to the parent directory of `mymodule` and execute these commands:

```
user@machine:~$ chmod +x pack.sh
user@machine:~$ ./pack.sh mymodule v4
Packing mymodule for platform v4 → mymodule.v4.tgz
Done.
```

It will create package with this content:

```
mymodule.v4.tgz
`-- mymodule.v4.tar
    |-- mymodule/
        |-- etc/
        |-- init
```

If you install this Router App in the router's GUI, you will see following messages in the Syslog:



```
[notice] mymodule: DEBUG: /opt/mymodule/etc/init defaults
[notice] mymodule: DEBUG: /opt/mymodule/etc/init start
[notice] umupdate: Module mymodule added.
[notice] https: user 'root' added user module 'mymodule.v4.tgz'
```

Here, the two rows are just comming from the `init` file (script).

7.3 Application Lifecycles

Shell scripts executed during Router App management (`install` , `uninstall` , `init`) should be designed to complete reasonably quickly (ideally within a few seconds) to avoid delaying the overall system operation or causing timeouts in the web interface.

Router App Installation Sequence

Installation occurs when a new Router App `*.tgz` archive is uploaded via the web interface (*Customization -> Router Apps*). The process is as follows:

1. User presses the *Add or Update* button and uploads the `*.tgz` archive.
2. The system extracts the archive and copies the contents to `/opt/mymodule` directory.
3. The install script is executed: `/opt/mymodule/etc/install`
4. The init script is called to set defaults: `/opt/mymodule/etc/init defaults` (This should create the `settings` file from `defaults`).
5. The init script is called to start the app: `/opt/mymodule/etc/init start`

Router App Update Sequence

Updating is triggered by uploading a `*.tgz` archive with the same name as an already installed Router App.

1. User presses the *Add or Update* button and uploads the new `*.tgz` archive.
2. The system identifies that an app with the same name exists.
3. The `init` script of the **currently installed (old) version** is called to stop it: `/opt/<RName>/etc/init stop`
4. The system automatically backs up the existing runtime configuration file: `/opt/<RName>/etc/settings` .
5. The system deletes all files and subdirectories of the old version from `/opt/<RName>` .
6. The system extracts the contents of the **new** `*.tgz` archive to `/opt/<RName>` .
7. The `install` script of the **new version** is executed (if present): `/opt/<RName>/etc/install` .

8. The `init` script of the **new version** is called to apply its defaults (if present):
`/opt/<RAname>/etc/init defaults` .
9. The system automatically restores the backed-up `settings` file from step 4, overwriting the `settings` file created in step 8. This merge operation ensures that previously configured values are retained for existing parameters, and new parameters get their defaults.
10. The `init` script of the **new version** is called to start the app (if present):
`/opt/<RAname>/etc/init start` .

Router App Uninstallation Sequence

Uninstallation is triggered by pressing the *Delete* button next to a Router App in the web interface.

1. User presses the *Delete* button for the target Router App.
2. The init script is called to stop the app: `/opt/mymodule/etc/init stop`
3. The uninstall script is executed: `/opt/mymodule/etc/uninstall` (Use this for any final cleanup).
4. The entire Router App directory (`/opt/mymodule`) and its contents are removed from the router's filesystem.

8. Building Router Apps

This chapter outlines the essential tools and resources required for developing compiled Router Apps in C or C++ for Advantech routers. Proper setup of the development environment is crucial for successful cross-compilation and deployment.

8.1 Overview of Development Tools

In addition to general development tools (such as a text editor and basic command-line familiarity, as mentioned in Chapter 6 *Getting Started with Router Apps*), creating compiled C/C++ Router Apps specifically requires:

- A **cross-compiler toolchain** tailored to the target Advantech router's architecture (e.g., ARMv7, AArch64). This toolchain allows you to compile code on your development machine (typically x86-64 Linux) that will run on the router.
- Optionally, but highly recommended, an **SDK (Software Development Kit)** provided by Advantech. The SDK often simplifies development by providing pre-configured build systems, example applications, helper libraries (like the `um` module for Python, or C equivalents), and documentation specific to Advantech platforms.

8.2 Cross-Compiler Toolchains

For compiling Router Apps written in C or C++, you must use a cross-compiler toolchain that matches the architecture of your target router platform (e.g., v2i, v3, v4, v4i). Advantech provides pre-built toolchains suitable for common development host environments.

The Advantech toolchains repository conveniently provides both Debian packages (`.deb`) for Debian-based Linux systems (such as Ubuntu) and RPM packages (`.rpm`) for RPM-based Linux distributions (such as Fedora or CentOS), allowing for straightforward installation on a wide range of development host systems.

Toolchain For	Router Platforms	Download Link
C/C++ Applications	v2i, v3, v4, v4i	https://bitbucket.org/bbsmartworx/toolchains

Table 2.: Advantech Cross-Compiler Toolchains

Always refer to the `README` file or documentation included within the downloaded toolchain archive for the most up-to-date installation and usage instructions specific to that toolchain version and your development host operating system.

Example: Installing Toolchains on a Debian-based Linux System

The following commands illustrate a typical process for cloning the repository and installing these packages:



```
# Clone the toolchains repository from Bitbucket
user@machine:~$ git clone https://bitbucket.org/bbsmartworx/toolchains.git
Cloning into 'toolchains'...
... (output from git clone) ...

# Navigate into the cloned directory
user@machine:~$ cd toolchains/

# Install all .deb packages found in the 'deb' subdirectory
# This command typically requires root privileges, hence 'sudo'.
user@machine:~/toolchains$ sudo dpkg -i deb/*.deb
... (output from dpkg -i, showing package installations) ...
```

After installing the toolchain packages, the cross-compilers (e.g., `armv7-linux-gnueabi-gcc` , `armv7-linux-gnueabi-g++` , `aarch64-linux-gnu-gcc` , `aarch64-linux-gnu-g++` , etc.) are typically installed under the `/opt/toolchain/` directory, for instance, in subdirectories like `/opt/toolchain/gcc-icr-v3-armv7-linux-gnueabi/bin/` or `/opt/toolchain/gcc-icr-v4-aarch64-linux-gnu/bin/` .

8.3 SDK (Software Development Kit)

Advantech provides an SDK (*ModulesSDK* on Bitbucket) designed to facilitate Router App development for both C/C++ and Python. Utilizing the SDK is highly recommended as it often includes:

- Example Router Apps demonstrating various functionalities and best practices.
- Helper libraries or modules (e.g., for interacting with router hardware like GPIOs, LEDs, or for web interface integration).
- Pre-configured Makefiles or build scripts that simplify the cross-compilation process for different router platforms.

Using the official SDK ensures better compatibility and access to platform-specific features.


Supported Languages	Router Platforms	SDK Download Link
C/C++ and Python	v2i, v3, v4, v4i	https://bitbucket.org/bbsmartworx/modulessdk

Table 3.: Advantech Router App SDK (ModulesSDK)

Consult the `README` file or other documentation included within the SDK for detailed instructions on its setup, structure, and usage.

Example: Cloning and Using the ModulesSDK

The following commands demonstrate cloning the *ModulesSDK* repository and typical steps for compiling libraries and example applications provided within it. Ensure that you have the appropriate cross-compiler toolchains (as described in Section [8.2 Cross-Compiler Toolchains](#)) installed and configured in your system's PATH before attempting to compile SDK examples.



```
# Clone the ModulesSDK repository, perhaps into a directory named 'ModulesSDK'
user@machine:~$ git clone https://bitbucket.org/bbsmartworx/modulesdk.git ModulesSDK
Cloning into 'ModulesSDK'...
... (output from git clone) ...

# Navigate into the cloned SDK directory
user@machine:~$ cd ModulesSDK/

# Compile common libraries provided by the SDK
user@machine:~/ModulesSDK$ make
... (output from make, compiling libraries) ...

# Clean all build artifacts
user@machine:~/ModulesSDK$ make clean
```

The exact `make` targets and variables (like `PLATFORM`) will depend on the specific structure and Makefiles within the *ModulesSDK*. Always refer to the SDK's documentation for precise build instructions.

If compiled successfully, the Router App **installation archives** are created in the `ModulesSDK/images` folder.

8.4 Building Your First Compiled Application with the SDK

This section provides a step-by-step guide to creating a simple command-line utility, `crc32sum`, as a compiled Router App using C and the Advantech ModulesSDK. The `crc32sum` utility will calculate the CRC32 (Cyclic Redundancy Check) checksum of files stored on the router. CRC32 is an error-detecting code commonly used to verify data integrity. This utility can be used by administrators or other scripts on the router to:

- Verify if a configuration file, firmware component, log file, or any other data file has been corrupted or unintentionally modified.
- Confirm data transfer success by comparing its CRC32 checksum against the source file's checksum.
- Be incorporated into shell scripts for automated integrity checks of critical files.

We will use the Advantech ModulesSDK for this example, as it is the recommended approach. The SDK simplifies the cross-compilation process and helps in building Router App installation packages (`*.tgz`) for various target router platforms. Ensure you have the ModulesSDK cloned and the necessary cross-compiler toolchains installed (refer to Chapter [8 Building Router Apps](#)).

Step 1: Writing the C Program

The C program, which we will name `crc32sum.c`, will take a filename as a command-line argument and print its CRC32 checksum.

Source code for `crc32sum.c`:

Place following code into `ModulesSDK/modules/crc32sum/source/crc32sum.c` file.

```

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h> // For uint32_t

// Standard CRC32 polynomial
#define CRC32_POLYNOMIAL 0xEDB88320L

// Function to calculate CRC32 (PKZIP, Ethernet, PNG standard)
static uint32_t calculate_crc32(FILE *file) {
    // Static table to ensure it's initialized only once
    static uint32_t crc_table[256];
    static int table_initialized = 0; // Flag to check if table is initialized
    uint32_t current_crc_value;
    int i, j;
    unsigned char byte_read;

    // Initialize CRC table only once
    if (!table_initialized) {
        uint32_t crc_entry;
        for (i = 0; i < 256; i++) {
            crc_entry = i;
            for (j = 0; j < 8; j++) {
                // LSB-first processing for table generation
                crc_entry = (crc_entry & 1) ? (crc_entry >> 1) ^ CRC32_POLYNOMIAL : crc_entry >> 1;
            }
            crc_table[i] = crc_entry;
        }
        table_initialized = 1;
    }

    // Calculate CRC of the file content
    current_crc_value = 0xFFFFFFFFL; // Initial CRC value (standard for this CRC32 variant)
    while (fread(&byte_read, 1, 1, file) == 1) {
        // LSB-first processing of byte into CRC
        current_crc_value = crc_table[(current_crc_value ^ byte_read) & 0xFF] ^ (current_crc_value >> 8);
    }
    return current_crc_value ^ 0xFFFFFFFFL; // Final XOR (standard for this CRC32 variant)
}

int main(int argc, char *argv[]) {
    FILE *file_ptr;
    uint32_t calculated_crc_value;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <filename>\n", argv[0]);
        return 1; // Indicate error
    }

    file_ptr = fopen(argv[1], "rb"); // Open in binary read mode
    if (file_ptr == NULL) {
        perror("Error opening file"); // perror prints the system error string
        return 1; // Indicate error
    }

    calculated_crc_value = calculate_crc32(file_ptr);
    fclose(file_ptr);
    printf("%08X\t%s\n", calculated_crc_value, argv[1]); // Output in uppercase hex
    return 0; // Indicate success
}

```

Key features of `crc32sum.c`:

- It includes `<stdint.h>` for fixed-width integer types like `uint32_t`.
- The `calculate_crc32` function implements a standard table-driven CRC32 algorithm, consistent with common CRC32/PKZIP variants.
- It reads the input file byte by byte in binary mode (`"rb"`).
- The `main` function handles command-line arguments (expecting a single filename) and file operations, including basic error checking.

- Output is formatted to print the 8-digit uppercase hexadecimal CRC32 value followed by a tab character and the filename, similar to utilities like `md5sum` .

Step 2: Creating the Router App Control Scripts and Metadata

This step involves preparing the necessary control scripts and metadata files within the `etc/` directory of your Router App. For this `crc32sum` utility, which is a command-line tool and not a background daemon, an `init` script for starting/stopping a service is not required. Instead, `install` and `uninstall` scripts will manage a symbolic link to make the command globally accessible.

The `install` script:

This script creates a symbolic link in `/usr/bin` , allowing `crc32sum` to be run without specifying its full path after the Router App is installed.

```
#!/bin/sh
# Create a symbolic link for crc32sum in /usr/bin

ln -sf /opt/crc32sum/bin/crc32sum /usr/bin/crc32sum

exit 0
```

The `uninstall` script:

This script removes the symbolic link created during installation when the Router App is uninstalled.

```
#!/bin/sh
# Remove the symbolic link for crc32sum from /usr/bin

rm -f /usr/bin/crc32sum

exit 0
```

Important: After creating these `install` and `uninstall` scripts, ensure they are made executable using the `chmod +x` command (e.g., `chmod +x install uninstall`) on your development machine before building the app.

Optional metadata files in `etc/` subdirectory:

- `name` : A plain text file containing the human-readable name for display in the router's web interface. Example content:

```
CRC32 Sum Utility
```

- `version` : A plain text file specifying the version of your Router App. Example content:


```
1.0.0 (2025-05-21)
```

Step 3: Preparing Makefiles for SDK Compilation

The Advantech ModulesSDK uses a Makefile-based build system. To integrate our `crc32sum` application:

1. **Create the module directory:** If it doesn't exist, create a directory for your new module within the SDK, for example: `ModulesSDK/modules/crc32sum/`.
2. **Copy the main Makefile template:** Copy the generic module `Makefile` from `ModulesSDK/modules/template/Makefile` to your new module's directory: `ModulesSDK/modules/crc32sum/Makefile`. This Makefile handles the overall process of building the module for different platforms and packaging it.
3. **Create the source Makefile:** Create a new directory `ModulesSDK/modules/crc32sum/source/`. Inside this `source/` directory, create another `Makefile` with the following content to define how your C code is compiled:

Content for ModulesSDK/modules/crc32sum/source/Makefile:



```
# Include common rules and definitions from the SDK's root
include ../../../../Rules.mk

# Define the name of the executable to be built
MODULE_TARGET_EXE = crc32sum
# Define the C source file(s) for the executable
MODULE_TARGET_SRC = crc32sum.c

# Optional: Add linker libraries if needed (e.g., -lm for math, -lpthread for pthreads)
# For crc32sum.c, no extra libraries are needed beyond standard C.
# LDLIBS += -lm

# Use a standard SDK macro to define the build rule for the program
$(eval $(call build-program, $(MODULE_TARGET_EXE), $(MODULE_TARGET_SRC)))

# Define the 'install' target for this source Makefile.
# This target is called by the main module Makefile to copy the compiled
# binary into the staging directory before packaging.
install:
    # Create the 'bin' directory in the staging area (DESTDIR) if it doesn't exist
    @install -d $(DESTDIR)/bin
    # Copy the compiled executable to DESTDIR/bin/ and set execute permissions (755)
    @install -m 755 $(OBJDIR)/$(MODULE_TARGET_EXE) $(DESTDIR)/bin/$(MODULE_TARGET_EXE)
```

Resulting Directory Structure for the `crc32sum` Module within SDK

After these steps, the directory structure for your `crc32sum` module within the ModulesSDK should look like this:

```
ModulesSDK/
|-- Rules.mk                      (and other root SDK files)
|-- modules/
|   |-- crc32sum/
|   |   |-- merge/
|   |   |   |-- etc/
|   |   |       |-- install      (install script)
|   |   |       |-- name        (metadata file)
|   |   |       |-- uninstall    (uninstall script)
|   |   |       |-- version      (metadata file)
|   |   |-- source/
|   |   |   |-- crc32sum.c      (your C source code)
|   |   |   |-- Makefile        (Makefile for compiling crc32sum.c)
|   |   |-- Makefile            (main Makefile for the crc32sum module, copied from template)
|   |-- template/
|   |   |-- Makefile            (original template Makefile)
|   |-- example1/
|   |-- example2/
|   |-- ...                    (other example modules)
|-- ...                        (other SDK directories like images/, libs/)
```

Step 4: Building the Router App Package

Once the source code and Makefiles are in place within the ModulesSDK structure:

- **Navigate to the SDK's root directory:**

```
user@machine:$ cd /path/to/your/ModulesSDK/
```

- **Build for all modules:**

Often, simply running `make` from the root of the SDK might build all modules for all default platforms if the main SDK Makefile is structured that way.

```
user@machine:/ModulesSDK$ make
```

The SDK's build system will handle the cross-compilation using the appropriate toolchain and then package the application into a `*.tgz` archive. The resulting installation packages are typically placed in a directory like `ModulesSDK/images/crc32sum/`. For instance, you might find

```
ModulesSDK/images/crc32sum/crc32sum.v4.tgz .
```


Step 5: Uploading and Testing

Upload the generated `*.tgz` archive file for the target platform (e.g., `crc32sum.v4.tgz`) to your Advantech router. This is typically done via the router's web interface, in the section *Customization* → *Router Apps*. After the Router App is installed by the system (which includes running your `install` script):

- The utility should be available as the `crc32sum` command globally due to the symbolic link created in `/usr/bin`.
- **Test from the router's CLI:**
 - Create a test file: `echo "test data" > /tmp/testfile.txt`
 - Run the utility: `crc32sum /tmp/testfile.txt`
 - Verify the output. For a file containing the exact string "test data" followed by a newline character (which `echo` typically adds), the CRC32 checksum should be `176BDC9D`. The output will appear as:
`176BDC9D /tmp/testfile.txt`

This example provides a compact C utility whose core logic (CRC32 calculation) is a strong candidate for C implementation rather than shell scripting. Using the SDK streamlines the build and packaging process for different router platforms.

8.5 Core Programming for Compiled Applications

Libraries and Dependency Management for Compiled Apps

To ensure the continued proper functioning of your Router App after router firmware updates, adhere to the following recommendations regarding libraries and dependencies:

- **Avoid dynamic linking** to libraries provided by the router's firmware, with the exception of the standard C library (`glibc`). Link other necessary libraries statically with your Router App whenever possible.
- **Do not rely on the presence or specific versions of other libraries** in the router's filesystem (e.g., in `/usr/lib`), except for `glibc`.

The reason for these recommendations is that libraries included in the router's firmware (other than `glibc`, which maintains strong backward compatibility guarantees) may change, be updated, or even be removed between different firmware versions or across router models. Making your Router App independent of these system libraries (except for relying on a baseline `glibc`) is crucial for ensuring it remains compatible across firmware updates and different router platforms.

If you are developing your Router App in C or C++, you can dynamically link against the `glibc` library provided by the router (typically located in `/lib` or `/usr/lib` via symlinks).

Interfacing with Router System Services and APIs

Compiled applications can interact with the router's system services and hardware through several mechanisms:

- **Standard Linux System Calls:** Direct invocation of system calls (e.g., `open()`, `read()`, `write()`, `socket()`, `fork()`) provides the most fundamental level of interaction with the Linux kernel and its services.
- **Shell Commands via `system()` or `popen()`:** Executing shell commands using the `system()` C library function or managing command I/O via `popen()`. This approach should be used with caution due to potential security risks (e.g., command injection if external input is used to construct commands without proper sanitization) and performance overhead compared to direct API calls or system calls.
- **Inter-Process Communication (IPC):** If your application needs to communicate with other daemons or processes running on the router, standard Linux IPC mechanisms such as pipes, FIFOs (named pipes), POSIX message queues, shared memory, or Unix domain sockets can be utilized.
- **Specific `ioctl` Calls for Hardware Access:** For direct hardware manipulation (e.g., GPIOs, LEDs, serial ports, or other custom hardware), `ioctl` system calls on specific device files (e.g., `/dev/gpiochip0`, `/dev/ttyS0`) are often the method of choice. The specific `ioctl` commands, request codes, and data structures are hardware and driver-dependent. Detailed information on these low-level hardware APIs should be available in the Advantech SDK documentation or platform-specific hardware/driver guides. The SDK often provides wrapper functions or libraries to simplify these interactions. (Note: The `ioctl` command-line utility is not typically present in BusyBox by default, so interaction is via the C system call.)

The Advantech SDK, when available for your target platform and development language, typically provides helper functions, libraries, and example code to simplify access to router-specific functionalities and hardware, abstracting some of the lower-level details and promoting more portable code across Advantech platforms.

Debugging and Testing Compiled Applications

Effective debugging and testing are crucial for developing robust compiled applications on an embedded platform like an Advantech router:

- **System Log Integration:** Utilize the `logger` command-line utility (if calling from shell scripts) or the `syslog()` C library functions (after `openlog()`) from within your C application to send diagnostic messages, status updates, and error reports to the router's system log. This is often the primary method for on-device debugging and monitoring. (Your console output confirms `logger` is available).
- **Debug Builds and Symbols:** When cross-compiling, create a debug version of your application by including debugging symbols (typically using the `-g` compiler flag with GCC). While on-router interactive debugging tools like GDB might be limited or not present in standard firmware, these symbols are invaluable if you can perform remote debugging (e.g., with GDB server if available) or analyze core dumps generated by application crashes.
- **Resource-Aware Testing:** Test your application thoroughly on the actual target router platform(s). Pay close attention to resource consumption (CPU utilization, RAM footprint, flash storage usage) under various operational conditions to ensure your app is efficient and does not destabilize the router.
- **SDK Examples and Documentation:** If an SDK is provided by Advantech for your router model, review its example applications and documentation. These can serve as valuable references for best practices, API usage, build system configuration, and platform-specific considerations.
- **Incremental Development and Unit Testing:** Develop and test components or features incrementally. For complex applications, start with basic functionality and add features step-by-step, testing at each stage. Where possible, write unit tests for individual functions or modules that can be run in your cross-compilation environment or on the target.
- **Robust Error Handling:** Implement comprehensive error handling in your C code. Diligently check return values of all system calls and library functions. Log errors appropriately (e.g., to syslog) to aid in diagnostics.
- **Cross-Platform Considerations:** If your Router App is intended to run on multiple Advantech router models or different firmware versions, test for compatibility and be mindful of potential differences in available libraries, kernel features, or hardware interfaces.

8.6 Developing Scripted Router Applications (Python)

Advantech routers support the development of Router Apps using Python, offering a high-level, rapidly developed alternative to compiled languages like C/C++ for many tasks. Python 3 is typically available on supported platforms via dedicated Python Router Apps (see Part [II Python](#) for general Python installation and usage on the router). This chapter focuses on leveraging Python specifically for creating Router App packages.

Python Environment within a Router App

When a Python script runs as part of an installed Router App, it operates within the Python environment provided by the active Python Router App (Full or Lite version) on the device. This environment includes the Python 3 interpreter and its standard library.

For router-specific functionalities, such as accessing GPIOs, reading system parameters, or generating HTML content for integration with the router's web interface, Advantech provides a special Python module named `um`. This module is included as part of an SDK. Always consult the SDK documentation for your target platform (refer to Section [8.3 SDK \(Software Development Kit\)](#)) for details on the availability and usage of the `um` module.

Advantages and Limitations of Python for Router Apps

Using Python for developing Router Apps presents several benefits and some considerations:

Advantages

- **Rapid Development:** Python's concise syntax and high-level nature generally lead to faster development cycles compared to C/C++.
- **Ease of Use:** Python is known for its readability and simpler learning curve, making it accessible for a wider range of developers.
- **Suitability for Specific Tasks:** Excellent for web scripting (e.g., CGI scripts for custom web interface pages), automation tasks (e.g., scheduled jobs, event-driven actions), data processing (e.g., parsing logs, manipulating configuration), and network scripting.
- **Rich Standard Library:** Python's extensive standard library provides many built-in modules for common tasks, reducing the need for external dependencies.
- **Third-Party Libraries (with Full Python RA):** The Full Python Router App, with `pip`, allows access to a vast ecosystem of third-party libraries via PyPI, greatly extending capabilities (see Section [4.5.1 Using `pip` to Install Third-Party Libraries](#)).

Limitations

- **Resource Consumption:** Python applications, being interpreted, can have higher CPU and memory overhead compared to equivalent applications compiled from C/C++. This is an important consideration for resource-constrained embedded routers.
- **Performance:** For extremely time-critical operations or computationally intensive tasks (e.g., high-speed packet processing, complex cryptographic calculations), the performance of Python might be a bottleneck compared to C/C++.

- **Dependency Management (Lite Python RA):** If using the Python Lite RA (which lacks `pip`), managing third-party dependencies requires manually bundling them with your Router App, which can be less convenient.
- **Startup Time:** Python scripts might have a slightly longer startup time compared to compiled binaries.

Application Structure for Python Router Apps

Python applications packaged as Router Apps follow the same general directory structure and packaging conventions as compiled Router Apps, as described in Chapter 7 *Router App Structure*. Key considerations for Python scripts include:

- **Executable Scripts and Daemons:** Python scripts intended to be run as main executables or background daemons (started by the `init` script of your Router App) should typically be placed in the `/bin` directory within your Router App package (e.g., `/opt/<app_name>/bin/myscript.py`). These scripts should have a shebang line (e.g., `#!/usr/bin/python3`) and be made executable (`chmod +x`).
- **Custom Python Modules:** If your application consists of multiple Python files or custom library modules, you can place them in the root directory of your Router App package (e.g., `/opt/<app_name>/mymodule.py`) or in a dedicated subdirectory (e.g., `/opt/<app_name>/lib/`). These modules can then be imported into your main scripts using standard Python import mechanisms (e.g., `import mymodule` or `from lib import my_utility`). Ensure the Python interpreter can find these modules (Python typically adds the script's own directory to `sys.path`).
- **CGI Scripts for Web Interface Integration:** Python scripts designed as CGI (Common Gateway Interface) applications to extend the router's web GUI must be placed in the `/www` subdirectory of your Router App package (e.g., `/opt/<app_name>/www/index.cgi`).
 - These CGI scripts must have execute permissions.
 - They must start with a shebang line pointing to the Python interpreter (e.g., `#!/usr/bin/python3`).
 - The first line of output from a CGI script must be a valid HTTP header, typically `Content-Type: text/html`, followed by a blank line, before any HTML content is printed. The `um` module, if used, often handles this.
- **Control Scripts (`init`, `install`, `uninstall`):** These scripts, located in `/etc`, are standard shell scripts (`#!/bin/sh`) and are used to manage the lifecycle of your Python Router App (e.g., starting/stopping a Python daemon). They are not Python scripts themselves.

Example: Python CGI for System Status Display

The following is an example (available in SDK as example7) of a Python CGI script, `index.cgi`, intended to be part of a Router App. It demonstrates how to use the (hypothetical or SDK-provided) `um` module to retrieve and display various system status information (like GPIO states, supply voltage, and internal temperature) on a custom web page within the router's web interface. This script would typically reside in `/opt/<app_name>/www/index.cgi` within your Router App package.

```
#!/usr/bin/python3

# *****
#
# CGI script of User Module
#
# Copyright (C) 2016-2024 Advantech Czech s.r.o.
#
# This Source Code Form is subject to the terms of the Mozilla Public
# License, v. 2.0. If a copy of the MPL was not distributed with this
# file, You can obtain one at http://mozilla.org/MPL/2.0/.
#
# *****

import um

MODULE_TITLE = b"Example 7"

um.html_page_begin(MODULE_TITLE)

um.html_form_begin(MODULE_TITLE, b"System Status", None, 0, None, None)

um.html_pre_head(b"Binary Input")

msg = b"Binary input <b>BIN0</b> is <b>"
msg += b"OFF" if um.gpio_get_bin0() else b"ON"
msg += b"</b>."

um.html_pre_text(msg)

um.html_pre_head(b"Binary Output")

msg = b"Binary output <b>OUT0</b> is <b>"
msg += b"ON" if um.gpio_get_out0() else b"OFF"
msg += b"</b>."

um.html_pre_text(msg)

um.html_pre_head(b"Supply Voltage")

voltage = (um.gpio_get_voltage() + 50) / 100

msg = "Supply voltage is <b>"
msg += str(voltage / 10.0) + " V" if voltage > 0 else "N/A"
msg += "</b>."

um.html_pre_text(msg.encode('utf-8'))

um.html_pre_head(b"Temperature")

temperature = um.gpio_get_temperature()

msg = "Internal temperature is <b>"
msg += str(temperature - 273) + " &degC" if temperature > 0 else "N/A"
msg += "</b>."

um.html_pre_text(msg.encode('utf-8'))

um.html_form_end(None)

um.html_page_end()
```

This example script is illustrative and demonstrates:


- Importing the router-specific `um` module (if available).
- Using functions from the `um` module (e.g., `um.html_page_begin()` , `um.gpio_get_bin0()`) to generate HTML content for display in the router's web interface and to retrieve hardware status information.
- The interpretation of values returned by hardware-access functions (e.g., whether 0 means ON or OFF for a binary input) is dependent on the specific implementation of the `um` module and the underlying hardware of the router model. This should be clearly documented by the SDK or module provider.
- Basic error handling using `try...except` blocks for robustness, especially when dealing with hardware interactions or external modules.
- The necessity of encoding Python strings to bytes (e.g., using `msg.encode('utf-8')`) if the `um` module's HTML functions expect byte strings.

The `um` module plays a crucial role in abstracting the low-level hardware and web interface details, making it simpler to access these features from Python scripts within a Router App.

Managing Dependencies for Python Router Apps

When your Python Router App relies on libraries:

- **Standard Libraries:** If your app uses only Python standard libraries, ensure they are part of the Python environment provided by the installed Python RA (Full or Lite) on the router. Most common standard libraries are usually included.
- **Third-Party Libraries (Not Pre-installed):**
 - **Full Python RA with `pip3`** : If the Full Python RA is installed, you can potentially use `pip3` to install required third-party libraries from PyPI directly onto the router as a post-installation step (e.g., triggered from your Router App's `install` or `init` script, or manually by an administrator). This requires internet access and careful consideration of storage space and compilation needs.
 - **Bundling Libraries:** For both Full and Lite RAs (especially mandatory for Lite), if libraries are not installable via `pip` or if you want a self-contained app, you must bundle the necessary pure-Python third-party libraries directly within your Router App package.
 - * Place the library's source code (e.g., the package directory) into a subdirectory within your Router App (e.g., `/opt/<app_name>/vendor/`).
 - * In your Python scripts, you may need to adjust `sys.path` at runtime to include this vendor directory so Python can find the bundled modules:



```
# In your main Python script:
import sys
import os
# Assuming your script is in /opt/<app_name>/bin/
# and vendor directory is /opt/<app_name>/vendor/
app_root = os.path.dirname(os.path.dirname(os.path.abspath(__file__)))
vendor_dir = os.path.join(app_root, "vendor")
if vendor_dir not in sys.path:
    sys.path.insert(0, vendor_dir)

# Now you can import your bundled library
# import my_bundled_library
```

* Alternatively, if your app structure allows, relative imports might be usable.

- **C Extensions:** Libraries with C extensions that require compilation are challenging to bundle directly without pre-compiling them for the target router architecture.
- **Virtual Environments (venv):** While `venv` is excellent for development, its direct usage and deployment within a Router App package on the router itself are generally less common due to increased storage footprint and complexity in managing activation for automated scripts. If used, the path to the virtual environment's interpreter must be explicitly invoked.

Deployment of Python Router Apps

Deployment of a Python-based Router App follows the same procedure as for any other Router App:

1. **Package Contents:** Package all necessary Python scripts (`.py`), custom modules, bundled third-party libraries (if any), CGI scripts, and any other required files (e.g., configuration templates, data files) into the `*.tgz` archive.
2. **Directory Structure:** Adhere to the standard Router App directory structure (e.g., `bin/` for executables, `www/` for CGI, `etc/` for control scripts, custom module directories).
3. **Permissions:** Ensure all scripts that need to be executed (main scripts in `bin/`, CGI scripts in `www/`) have execute permissions set (`chmod +x`) before packaging. The control scripts in `etc/` (`init`, `install`, `uninstall`) must also be executable shell scripts.
4. **Installation:** Upload the `*.tgz` package to the router via the web interface (*Customization* → *Router Apps* or similar menu). The router's system will then handle the extraction and setup, including running your `install` script.

Refer to Chapter 7 *Router App Structure* for general Router App packaging and lifecycle details.

9. Summary and Best Practices

9.1 Key Development Constraints Recap

- Storage space in the `/opt` directory, where Router Apps are installed, is limited. Refer to Table 9 for partition sizes specific to each platform. Plan your application size accordingly.
- On platforms with a 128 KiB MRAM partition for `/var/data` (e.g., standard v3), limit your Router App's usage of this space to approximately 64 KiB to ensure sufficient space for the operating system. See Section 15.1 for details.
- Note that the `/opt` directory persists across router firmware updates, preserving installed Router Apps. Data stored in `/var/data` also generally persists.
- Router Apps run within a BusyBox environment, which provides a subset of standard Linux commands and shell features. Ensure your scripts use POSIX-compliant shell syntax (`#!/bin/sh`) and rely only on commands available on the target router.
- Be mindful of RAM and CPU limitations detailed in Chapter 15.
- Adhere to library linking guidelines (Section 8.5) for C/C++ apps to maintain firmware compatibility.
- Follow security best practices for web interfaces.

9.2 Best Practices Recap

- Use the official SDKs and toolchains when available.
- Structure your application package correctly (`*.tgz` with a single top-level directory).
- Implement robust `init` , `install` , and `uninstall` scripts.
- Manage application configuration using `defaults` and `settings` files.
- Keep scripts and applications lightweight and efficient.
- Test thoroughly on target hardware and across different firmware versions if aiming for broad compatibility.
- Use logging for easier debugging.
- Secure web interfaces appropriately.

9.3 Firewall Rules for Router Apps

If your Router App runs a server accepting connections on a specific TCP or UDP port, you must configure the router's firewall (`iptables`) to allow this traffic, especially if the router's "Default Server" NAT option is enabled.



The *Send all remaining incoming packets to default server* option in the NAT configuration (see Figure 2) can redirect incoming traffic to a specified internal IP address if no other NAT or firewall rule explicitly handles it. This can prevent connections from reaching your Router App.

To ensure traffic reaches your application, the Router App should manage its own firewall rules, typically within its `init` script (`/opt/<RName>/etc/init`), adding rules on `start` and removing them on `stop` .

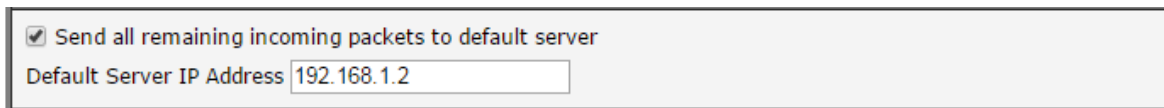


Figure 2.: The Default Server Option in the NAT Configuration (for IPv4)

Integration with Router Firewall Chains: Advantech router firmware often provides predefined chains designed for Router App firewall rules to integrate cleanly with the system's overall firewall structure. Use these chains:

- `in_mod chain` (`filter` table): This chain is typically jumped to from the main `INPUT` chain (which filters traffic destined *for the router itself*). Rules placed here (or in a custom chain jumped to from here) can explicitly `ACCEPT` traffic destined for the port your application is listening on.
- `pre_mod chain` (`nat` table): This chain is typically jumped to early in the `PREROUTING` chain (which handles incoming packets *before* the routing decision, primarily for DNAT). By adding a rule here (or in a custom chain jumped to from here) that `ACCEPT` s traffic destined for your application's port, you prevent that traffic from being processed by later DNAT rules in the `PREROUTING` chain, specifically bypassing the "Default Server" rule.

The example `init` script snippet below shows functions `add_chain()` and `del_chain()` for managing these rules using a unique chain for the application. The `add_chain()` function is called during `init start` with parameters like `mod_mymodule tcp 1000` , where `mod_mymodule` is a unique identifier (custom chain name) for the app's rules, `tcp` is the protocol, and `1000` is the port number (which should ideally be read from the app's settings file). The `del_chain()` function, called during `init stop` , removes the custom chain and the jumps to it, cleaning up the firewall rules. The `del_chain` function in the example includes checks to prevent errors if rules or chains don't exist before attempting deletion.

Example Functions for init Script:

```

MODNAME=mymodule
MODEXEC=mymoduled
add_chain() {
    /sbin/iptables -N $1 || return
    /sbin/iptables -A $1 -p $2 --dport $3 -j ACCEPT
    /sbin/iptables -A in_mod -j $1
    /sbin/iptables -t nat -N $1
    /sbin/iptables -t nat -A $1 -p $2 --dport $3 -j ACCEPT
    /sbin/iptables -t nat -A pre_mod -j $1
    if [ -f /sbin/ip6tables ]; then
        /sbin/ip6tables -N $1 || return
        /sbin/ip6tables -A $1 -p $2 --dport $3 -j ACCEPT
        /sbin/ip6tables -A in_mod -j $1
        /sbin/ip6tables -t nat -N $1
        /sbin/ip6tables -t nat -A $1 -p $2 --dport $3 -j ACCEPT
        /sbin/ip6tables -t nat -A pre_mod -j $1
    fi
}
del_chain() {
    # Check if chain exists before attempting to delete
    /sbin/iptables -L $1 >/dev/null 2>&1
    if [ $? -eq 0 ]; then
        /sbin/iptables -D in_mod -j $1 2>/dev/null
        /sbin/iptables -F $1
        /sbin/iptables -X $1
    fi
    # Check if nat chain exists
    /sbin/iptables -t nat -L $1 >/dev/null 2>&1
    if [ $? -eq 0 ]; then
        /sbin/iptables -t nat -D pre_mod -j $1 2>/dev/null
        /sbin/iptables -t nat -F $1
        /sbin/iptables -t nat -X $1
    fi
    # Handle IPv6 rules if ip6tables exists
    if [ -f /sbin/ip6tables ]; then
        /sbin/ip6tables -L $1 >/dev/null 2>&1
        if [ $? -eq 0 ]; then
            /sbin/ip6tables -D in_mod -j $1 2>/dev/null
            /sbin/ip6tables -F $1
            /sbin/ip6tables -X $1
        fi
        /sbin/ip6tables -t nat -L $1 >/dev/null 2>&1
        if [ $? -eq 0 ]; then
            /sbin/ip6tables -t nat -D pre_mod -j $1 2>/dev/null
            /sbin/ip6tables -t nat -F $1
            /sbin/ip6tables -t nat -X $1
        fi
    fi
}

```



Example case Statement in init Script:

```

case "$1" in
    start)
        echo -n "Starting module $MODNAME: "
        . /opt/$MODNAME/etc/settings
        [ "$MOD_EXAMPLE5_ENABLED" != "1" ] && echo "skipped" && exit 0
        add_chain mod_$MODNAME tcp $MOD_MYMODULE_PORT 2> /dev/null
        /opt/$MODNAME/bin/$MODEXEC &
        RETVAL=$?
        [ $RETVAL = 0 ] && echo "done" || echo "failed"
        exit $RETVAL
        ;;
    stop)
        echo -n "Stopping module $MODNAME: "
        killall $MODEXEC 2> /dev/null
        del_chain mod_$MODNAME 2> /dev/null
        RETVAL=$?
        [ $RETVAL = 0 ] && echo "done" || echo "failed"
        exit $RETVAL
        ;;
    *)
        echo "Usage: $0 {start|stop|restart|status|defaults}"
        exit 1
esac

```

Simplified iptables Structure Overview: The following illustrates where the custom module chains (`mod_...` , created by `add_chain`) typically fit into the router's packet processing flow.

- **nat Table (Connection Tracking, Address/Port Translation)**
 - PREROUTING Chain (Incoming packets, before routing decision)
 - * ... (Standard rules)
 - * JUMP to `pre` chain (Typically for WAN interfaces). This chain usually jumps to `pre_mod` , which contains specific modification rules such as:
 - JUMP to `mod_app1` chain (App 1's rules)
 - JUMP to `mod_app2` chain (App 2's rules)
 - ... (Further rules added by `add_chain`)
 - * ... (Standard DNAT rules, potentially Default Server DNAT rule)
 - POSTROUTING Chain (Outgoing packets, after routing decision)
 - * ...
 - * ... (Standard SNAT/MASQUERADE rules)

By adding an `ACCEPT` rule in a custom chain jumped to from `pre_mod` (nat table), the Router App prevents its traffic from being redirected by later DNAT or Default Server rules. The corresponding `ACCEPT` rule jumped to from `in_mod` (filter table) explicitly allows the traffic to reach the application process running on the router.

Part IV.

Controlling Router Peripherals

10. Digital Input/Output Interfaces

10.1 io Utility

The `io` utility allows controlling binary outputs and reading binary/analog/counter inputs from the command line.



Binary I/O often uses inverse logic. Active physical input (e.g., high voltage) might read as logical `0`. Setting output to `1` might result in a low voltage physically. Always consult the specific router model's User Manual for I/O logic details.

Synopsis

`io get <pin>` or `io set <pin> <value>`

Command	Description
<code>get <pin></code>	Reads the state of input <pin> (e.g., <code>bin0</code> , <code>an1</code> , <code>cnt1</code>).
<code>set <pin> <value></code>	Sets the state of output <pin> to <value> (typically 0 or 1).

Table 4.: `io` Utility Commands

Pin Names

Refer to the router's User Manual for available pin names (`bin0`, `out0`, `an1`, `cnt1`, etc.), which depend on the model and installed expansion modules (e.g., XC-CNT).

Examples

- `io set out0 1` : Sets binary output OUT0 to state 1.
- `io get bin0` : Reads the state of binary input BIN0. Check exit code `$?` (0 or 1).
- `io get an1` : Reads the value of analog input AN1 (if XC-CNT present).
- `io get cnt1` : Reads the value of counter input CNT1 (if XC-CNT present).

10.2 Activate Binary Output via SMS



See Section [3.1 Handling Incoming SMS with a Custom Script](#) for details on the custom SMS handling mechanism using `/var/scripts/sms`. Ensure this mechanism is enabled in the router's SMS configuration.

This example demonstrates the implementation of a new SMS command, "IMPULSE", which activates binary output OUT0 for 5 seconds. It is triggered when an SMS containing the text "IMPULSE" is received by the router. The command is processed only if the sender is authorized.

Authorization Logic Options

The example script includes two common ways to authorize the sender:

1. Check the flag passed by the system (`$1`): If the sender's number is listed in the *Phone Number x* fields in the router's SMS settings GUI, `$1` will be `1`.
2. Hardcode specific phone number(s) directly in the script and compare against the sender's number (`$2`).

The example uses a combination (OR logic). Adjust the authorization check as needed for your requirements.

Startup Script

This script creates the `/var/scripts/sms` handler script in RAM at boot time.



```
#!/bin/sh

# Create the SMS handler script in RAM
cat > /var/scripts/sms << EOF
#!/bin/sh

# Specify the authorized phone number
PHONE=+420123456789

if [ "$1" = "1" ] || [ "$2" = "$PHONE" ]; then
    if [ "$3" = "IMPULSE" ]; then
        io set out0 1
        sleep 5
        io set out0 0
    fi
fi
EOF
```

How It Works

- The startup script creates the handler script `/var/scripts/sms`.
- Inside the handler script:

- It first checks for authorization:
 - * `["\$1" = "1"]` : Is it an authorized sender?; `||` : Logical OR.
 - * `["\$2" = "$PHONE"]` : Is the sender's phone number hardcoded?
- `if ["\$3" = "IMPULSE"]` : Checks if the third parameter (the first word of the SMS text) is exactly "IMPULSE". Note: This check is case-sensitive.
- If the command matches:
 - * `io set out0 1` : Uses the Advantech `io` utility to set binary output OUT0 to state 1 (check router manual for physical logic - often active-low).
 - * `sleep 5` : Pauses execution for 5 seconds.
 - * `io set out0 0` : Sets binary output OUT0 back to state 0 (inactive state).
- This sequence effectively creates a 5-second pulse on OUT0 when an authorized SMS containing the word "IMPULSE" is received.

10.3 Send Email on Binary Input Activation



Make sure you have correctly configured the SMTP server in *Configuration* → *Services* → *SMTP*. Refer to Chapter 3.2 *Email Configuration Notes* if applicable.

This script sends an informational email when the state of binary input BIN0 changes to active. It continuously monitors the input and sends a notification only when the input transitions from inactive (state 1) to active (state 0, assuming active-low logic).

Startup Script

```
#!/bin/sh

# Specify email address
EMAIL=john.doe@email.com

# Specify email subject
MESSAGE="BIN0 is active"

while true
do
    io get bin0
    VAL=$?
    if [ "$VAL" != "$OLD" ]; then
        [ "$VAL" = "0" ] && email -t $EMAIL -s "$MESSAGE"
        OLD=$VAL
    fi
    sleep 1
done
```



How It Works

- The script defines the destination email address (`EMAIL`) and the email subject (`MESSAGE`).
- It enters an infinite loop (`while true`) to continuously monitor the input.
- Inside the loop:
 - `io get bin0` : Reads the current state of binary input BIN0 using the `io` utility.
 - `VAL=$?` : Captures the exit status of `io get` . For binary inputs, this is typically 0 for the active state and 1 for the inactive state (verify with router manual).
 - `if ["$VAL" != "$OLD"]` : Checks if the current state (`VAL`) is different from the state stored from the previous iteration (`OLD`). This detects any change.
 - `["$VAL" = "0"] && email -t $EMAIL -s "$MESSAGE"` : This is a short-circuit AND condition executed *only if the state has changed*. If the new state (`VAL`) is 0 (active), it executes the `email` command to send the notification.
 - `OLD=$VAL` : Updates the stored state (`OLD`) with the current state (`VAL`) for the next loop iteration.
 - `sleep 1` : Pauses the script for 1 second before checking the input again.
- This logic ensures an email is sent only upon the transition from inactive (1) to active (0), avoiding repeated emails if the input remains active or becomes inactive.

Python Script

Here is an equivalent script in Python. Save the code below into a file named `send_email.py`.



```
#!/usr/bin/python3
import sys
import time
import subprocess

# Configuration
EMAIL_RECIPIENT = "john.doe@email.com"
EMAIL_SUBJECT = "BINO is active"
INPUT_PIN = "bin0" # The binary input pin to monitor
POLL_INTERVAL_SECONDS = 1
ACTIVE_STATE_EXIT_CODE = 0

def get_input_pin_state(pin_name):
    try:
        command = ["io", "get", pin_name]
        process = subprocess.run(command, capture_output=True, text=True)
        return process.returncode
    except FileNotFoundError:
        print(f"Error: The 'io' command was not found. Please ensure it's in the PATH.", file=sys.stderr)
        return None
    except Exception as e:
        print(f"An unexpected error occurred while getting pin {pin_name} state: {e}", file=sys.stderr)
        return None

def send_email_notification(recipient, subject):
    try:
        command = ["email", "-t", recipient, "-s", subject]
        subprocess.run(command, check=True, capture_output=True, text=True)
        print(f"Email notification sent to {recipient} with subject: {subject}", file=sys.stderr)
    except FileNotFoundError:
        print(f"Error: The 'email' command was not found. Please ensure it's in the PATH.", file=sys.stderr)
    except subprocess.CalledProcessError as e:
        print(f"Error executing 'email' command: {e.stderr.strip()}", file=sys.stderr)
    except Exception as e:
        print(f"An unexpected error occurred while sending email: {e}", file=sys.stderr)

def main():
    old_state = None

    print(f"Monitoring {INPUT_PIN} every {POLL_INTERVAL_SECONDS} second(s). Press Ctrl+C to stop.", file=sys.stderr)

    try:
        while True:
            current_state = get_input_pin_state(INPUT_PIN)

            if current_state is not None:
                if current_state != old_state:
                    if current_state == ACTIVE_STATE_EXIT_CODE:
                        print(f"Input {INPUT_PIN} transitioned to ACTIVE. Sending email.", file=sys.stderr)
                        send_email_notification(EMAIL_RECIPIENT, EMAIL_SUBJECT)
                        old_state = current_state
                    time.sleep(POLL_INTERVAL_SECONDS)
    except KeyboardInterrupt:
        print("\nMonitoring stopped by user.", file=sys.stderr)
    except Exception as e:
        print(f"An unexpected error occurred in the main loop: {e}", file=sys.stderr)

if __name__ == "__main__":
    main()
```

How It Works

This Python script continuously monitors a specified binary input pin on the router. If the pin transitions to an active state, the script sends an email notification.

- **Initialization and Configuration:**

- The script starts with a shebang line (`#!/usr/bin/python3`), ensuring it's executed by the Python 3 interpreter.
- It imports standard Python modules: `sys` (for system interaction, though minimally used here beyond `sys.stderr`), `time` (for pausing execution), and `subprocess` (for running external router commands like `io` and `email`).
- Essential configuration parameters are defined at the beginning:
 - * `EMAIL_RECIPIENT` : The email address to which notifications will be sent.
 - * `EMAIL_SUBJECT` : The subject line for the notification email.
 - * `INPUT_PIN` : The name of the binary input pin to be monitored (e.g., "bin0").
 - * `POLL_INTERVAL_SECONDS` : The duration in seconds the script waits between checks of the input pin's state.
 - * `ACTIVE_STATE_EXIT_CODE` : Defines the exit code returned by the `io get <pin_name>` command that signifies the pin is in its active state (typically 0 for Advantech routers).

- **Reading Input Pin State (`get_input_pin_state` function):**

- This function is responsible for querying the current state of the specified binary input pin.
- It constructs and executes the router's command `io get <pin_name>` using `subprocess.run()` .
- The function returns the exit code of the `io get` command. For Advantech router binary inputs, an exit code of 0 usually indicates an active state, and 1 indicates an inactive state.
- It includes error handling: if the `io` command is not found or another error occurs during execution, it prints an error message to standard error (`sys.stderr`) and returns `None` .

- **Sending Email Notification (`send_email_notification` function):**

- This function handles the sending of email alerts.
- It takes the recipient email address and the email subject as arguments.
- It forms and executes the router's command `email -t <recipient> -s <subject>` using `subprocess.run()` . This relies on the router having a configured `email` utility.
- Error handling is included to catch issues such as the `email` command not being found or the command failing (e.g., mail server misconfiguration), printing error messages to `sys.stderr` .

- **Main Monitoring Loop (`main` function):**

- A variable `old_state` is initialized to `None` , representing an unknown initial state of the input pin.
- A startup message is printed to `sys.stderr` , indicating which pin is being monitored and the polling interval.
- The script enters an infinite loop (`while True:`) to continuously monitor the input pin. This loop can be interrupted by pressing Ctrl+C (`KeyboardInterrupt`).
- **Inside the loop:**

- * `current_state = get_input_pin_state(INPUT_PIN)` : The current state of the input pin is read.
- * `if current_state is not None:` : The script only proceeds if the pin state was successfully read (i.e., `io.get` didn't fail).
- * `if current_state != old_state:` : This condition checks if the pin's state has changed since the last check. This is key to sending an email only upon a state transition.
 - `if current_state == ACTIVE_STATE_EXIT_CODE:` : If the state has changed *and* the new state is the defined active state (e.g., exit code 0), an informational message is printed to `sys.stderr`, and the `send_email_notification` function is called.
 - `old_state = current_state` : After processing any change, `old_state` is updated to the `current_state` for the next iteration. This ensures that an email is sent only once when the pin transitions to active, and not repeatedly if it remains active.
- * `time.sleep(POLL_INTERVAL_SECONDS)` : The script pauses for the specified interval before repeating the loop.
- The main loop is wrapped in a `try...except KeyboardInterrupt` block to allow the user to stop the script gracefully. A general `except Exception` block is also present to catch other unexpected errors during the loop's execution.

• Script Execution Entry Point:

- The standard `if __name__ == "__main__":` construct ensures that the `main()` function is called only when the script is executed directly.

Script Testing

Below is the console output, displaying the script's creation (using `vi`) and its testing.

```
/var/scripts # vi send_email.py
(...create and save script with pi...)
/var/scripts # chmod +x send_email.py
/var/scripts # ./send_email.py
Monitoring bin0 every 1 second(s). Press Ctrl+C to stop.
Input bin0 transitioned to ACTIVE. Sending email.
Email notification sent to x.y@advantech.com with subject: BIN0 is active
^C
Monitoring stopped by user.
/var/scripts #
```

10.4 Send SNMP Trap on Binary Input State Change



Make sure you have correctly configured the SNMP manager in *Configuration* → *Services* → *SNMP*.

This script sends an SNMP trap to the configured SNMP manager whenever the state of binary input BIN0 changes (either becoming active or inactive). It continuously monitors the input state.

Startup Script

```
#!/bin/sh

# Specify SNMP manager address
SNMP_MANAGER=192.168.1.2

while true
do
    io get bin0
    VAL=$?
    if [ "$VAL" != "$OLD" ]; then
        snmptrap $SNMP_MANAGER 1.3.6.1.4.1.30140.2.3.1.0 u $VAL
        OLD=$VAL
    fi
    sleep 1
done
```

How It Works

- The script defines the IP address of the SNMP manager (`SNMP_MANAGER`).
- It enters an infinite loop (`while true`) for continuous monitoring.
- Inside the loop:
 - `io get bin0` : Reads the current state of binary input BIN0.
 - `VAL=$?` : Captures the exit status (state: 0=active, 1=inactive).
 - `if ["$VAL" != "$OLD"]` : Checks if the state has changed since the last check.
 - If the state has changed:
 - * `snmptrap $SNMP_MANAGER 1.3.6.1.4.1.30140.2.3.1.0 u $VAL` : Sends an SNMP trap.
 - `$SNMP_MANAGER` : The destination IP address.
 - `1.3.6.1.4.1.30140.2.3.1.0` : The specific OID (Object Identifier) being sent. This OID represents the state of `bin0` .
 - `u` : Specifies the data type of the value being sent as Unsigned32.
 - `$VAL` : The current state (0 or 1) of the input pin.
 - * `OLD=$VAL` : Updates the stored state for the next comparison.
 - `sleep 1` : Pauses for 1 second before the next check.
- This script sends an SNMP trap containing the specific OID and the new state (0 or 1) every time the binary input changes state.


11. Serial Interfaces

This chapter provides an overview of using serial interfaces on Advantech routers. It covers identifying available serial ports, command-line utilities for their configuration and use, and an example of interacting with a serial port using a Python script packaged as a Router App via the Advantech ModulesSDK.

11.1 Identifying Serial Interfaces

Advantech routers support various serial interface standards, with RS-232 and RS-485 being common. The specific interfaces available (e.g., physical DB9 ports, terminal blocks, or those provided via expansion modules like PORT1/PORT2) and their parameters (e.g., dedicated ttyS* device, configurable modes) vary by router model. Always consult the hardware manual for your specific router model for detailed specifications.

To list available serial device nodes on a router, you can use the following console command. The presence of a device node in `/dev/` (e.g., `/dev/ttyS0`, `/dev/ttyUSB0`) indicates that the kernel recognizes a serial interface. However, this does not always mean a physical port is directly accessible on the device exterior without additional configuration or hardware. Serial interfaces can also be provided by USB-to-UART converters, which typically appear as `/dev/ttyUSB*` devices.



```
~ # ls -l /dev/tty*
crw-rw-rw-  1 root    root      5,   0 Jan  1  1970 /dev/tty
crw-----  1 root    root     249,  0 Jan  1  1970 /dev/ttyS0
crw-rw----  1 root    daemons 249,  1 Jan  1  1970 /dev/ttyS1
crw-rw----  1 root    daemons 249,  5 Jan  1  1970 /dev/ttyS5
crw-rw----  1 root    daemons 188,  0 May 26 06:56 /dev/ttyUSB0
crw-rw----  1 root    daemons 188,  1 May 26 06:56 /dev/ttyUSB1
% ... (other ttyUSB* devices if present) ...
```

Serial port configuration (baud rate, data bits, parity, stop bits) can be managed using command-line utilities or programmatically, as detailed in the following sections.

11.2 Command-Line Utilities for Serial Ports

Two common utilities for managing serial ports from the command line are `stty` and `portd`.

stty - Set and Print Terminal Line Settings

The `stty` program is used to change and print terminal line settings, including those for serial ports. It allows you to configure parameters like baud rate, character size, parity, stop bits, and flow control.

Synopsis:

```
stty [-a|g] [-F DEVICE] [SETTING]...
```

Common Options:

Option	Description
-F DEVICE	Open and use the specified DEVICE instead of standard input.
-a	Print all current settings in human-readable form.
-g	Print all current settings in a stty-readable form (can be used to save and restore settings).
[SETTING] ...	One or more settings to apply. Common settings include: <ul style="list-style-type: none"> ◦ <N>: Set speed to N bits per second (e.g., 115200). ◦ cs<N>: Set character size to N bits (cs7 or cs8). ◦ cstopb: Use two stop bits (prefix with - for one stop bit, e.g., -cstopb). ◦ parenb: Enable parity generation/detection. ◦ -parodd: Use even parity (if parenb is set). parodd for odd parity. ◦ -inpck: Disable input parity checking. ◦ ignpar: Ignore characters with parity errors. ◦ [-]raw: Enable (or disable with -) raw input. Disables most input processing. ◦ [-]echo: Enable (or disable) echoing of input characters. ◦ [-]crtcts: Enable (or disable) RTS/CTS hardware flow control. ◦ [-]ixon: Enable (or disable) XON/XOFF software flow control. For a full list, consult the stty man page or BusyBox documentation.

Table 5.: Common stty Options and Settings.

**Examples:**

To display all current settings for the serial port `/dev/ttyS0` :



```
stty -F /dev/ttyS0 -a
```

To display only the current speed of `/dev/ttyS1` :



```
stty -F /dev/ttyS1 speed
```

To configure `/dev/ttyS0` to 115200 bps, 8 data bits, no parity, 1 stop bit (8N1), and enable raw mode:



```
stty -F /dev/ttyS0 115200 cs8 -cstopb -parenb raw
```

portd - Serial Port to TCP/UDP Redirector

The `portd` daemon is a utility that provides transparent data transfer between a serial line and a TCP or UDP network connection. It can operate either as a server (listening for incoming network connections and forwarding data to/from the serial port) or as a client (connecting to a remote network host and forwarding data). This is often used for "Serial-to-Ethernet" or "Serial-over-IP" applications.

Synopsis:

```
portd -c <device> [-b <baudrate>] [-d < databits>] [-p <parity>] [-s <stopbits>]  
      [-l <split timeout>] [-4] [-h <hostname>] [-o <proto>] -t <port>  
      [-k <keepalive time>] [-i <keepalive interval>] [-j <inactivity timeout>]  
      [-n <reject new>] [-r <keepalive probes>] [-u user] [-x] [-z] [-f]
```


Supported Options:

Option	Description
-c <device>	Required. Serial line device (e.g., /dev/ttyS0).
-b <baudrate>	Baud rate (e.g., 115200). Default typically 9600.
-d <databits>	Number of data bits (7 or 8). Default typically 8.
-p <parity>	Parity: none, even, odd. Default typically none.
-s <stopbits>	Number of stop bits (1 or 2). Default typically 1.
-l <split timeout>	Data split timeout in milliseconds. If no data arrives from the serial port for this timeout, buffered data is sent over the network. Default typically 50.
-4	Forced detection for RS-485 on an Expansion Port. (Advantech-specific functionality).
-h <hostname>	Remote hostname or IP address to connect to (client mode). If not specified, portd runs in server mode.
-o <proto>	Network protocol: tcp or udp.
-t <port>	Required. TCP or UDP port number.
-k <keepalive time>	TCP Keepalive: Time (seconds) of inactivity before sending the first keepalive probe. Default: disabled.
-i <keepalive intvl>	TCP Keepalive: Interval (seconds) between subsequent keepalive probes. (Often named <keepalive interval> in help).
-j <inactivity to>	Inactivity timeout in seconds. If no data is transferred on the network connection for this period, the connection might be closed. (Often named <inactivity timeout> in help).
-n <reject new>	When acting as a server and a connection limit is reached (e.g., typically 1 client by default if -N is not supported or set), this option might control how new incoming connection attempts are handled (e.g., reject immediately). The exact behavior should be verified.
-r <keepalive probes>	TCP Keepalive: Number of unacknowledged probes before considering the connection dead.
-u <user>	Run portd as a specified user after starting (drops root privileges if started as root).
-x	Use CD (Carrier Detect) line as an indicator of TCP connection status (server mode).
-z	Use DTR (Data Terminal Ready) line to control/reflect TCP connection status (server mode).
-f	Enable flow control. The type of flow control (hardware/software) might be auto-detected or a default. For specific control (e.g., RTS/CTS vs XON/XOFF), underlying system settings via stty might be needed if portd doesn't offer finer granularity.

Table 6.: Supported portd options based on router's help output.

**Examples:**

To run `portd` as a TCP server on port 1000, redirecting data to/from `/dev/ttyS0` configured at 115200 bps, 8 data bits, no parity, 1 stop bit, and with flow control enabled, running it in the background:



```
portd -c /dev/ttyS0 -b 115200 -d 8 -p none -s 1 -f -o tcp -t 1000 &
```

To run `portd` as a TCP client, connecting to 192.168.1.100 on port 2000, forwarding data from `/dev/ttyS1` (9600 bps, 8N1), and setting an inactivity timeout of 300 seconds:



```
portd -c /dev/ttyS1 -b 9600 -h 192.168.1.100 -o tcp -t 2000 -j 300 &
```

11.3 Scripting Serial Communication with the `um` Python Module

This example demonstrates how to use the Advantech `um` Python module to communicate over a serial port (e.g., `/dev/ttyS0`) on the router. The Python script will be packaged as a Router App using the ModulesSDK.



To run Python scripts on the router, the *Python 3* or *Python 3 Lite* Router App must be installed from the router's web interface (*Customization* → *Router Apps*) or be part of the firmware.

Step 1: Writing the Python Script

We will create a simple Python script named `serial_um_example.py`. This script will open `/dev/ttyS0`, send a command, attempt to read a response, and then print the response. For this example to fully work, a device capable of responding must be connected to `/dev/ttyS0` and configured with matching serial parameters.

Source code for `serial_um_example.py`:

This file will be placed in `ModulesSDK/modules/serial_um_example/source/serial_um_example.py`.



```
#!/usr/bin/python3
import um
import sys # For exiting with error code

# Define serial port parameters
SERIAL_DEVICE = b"/dev/ttyS0"
BAUD_RATE = 115200
DATA_BITS = 8
PARITY = b"N" # None
STOP_BITS = 1

# Command to send and timeout for response
COMMAND_TO_SEND = b"ATI\r\n" # Example: AT command to request modem info
RESPONSE_TIMEOUT_SEC = 5

def main():
    print(f"Attempting to open serial port: {SERIAL_DEVICE.decode()} at {BAUD_RATE} bps...")
    fd = um.com_open(SERIAL_DEVICE, BAUD_RATE, DATA_BITS, PARITY, STOP_BITS)

    if fd < 0:
        print(f"Error: Failed to open serial port {SERIAL_DEVICE.decode()}. Error code: {fd}")
        sys.exit(1)

    print(f"Serial port opened successfully (fd: {fd}). Sending command...")

    try:
        received_data = um.com_xmit(fd, COMMAND_TO_SEND, RESPONSE_TIMEOUT_SEC)

        print(f"Sent to {SERIAL_DEVICE.decode()}: {COMMAND_TO_SEND.decode(errors='replace').strip()}")

        if received_data:
            print(f"Received from {SERIAL_DEVICE.decode()}:")
            # Print byte-by-byte hex and ASCII for detailed view if needed
            # print("Hex: " + " ".join(f"{b:02x}" for b in received_data))
            print(f"ASCII: {received_data.decode(errors='replace')}")
        else:
            print("No data received within the timeout period.")

    except Exception as e:
        print(f"An error occurred during serial communication: {e}")
    finally:
        print(f"Closing serial port (fd: {fd})...")
        um.com_close(fd)
        print("Serial port closed.")

if __name__ == "__main__":
    main()
```

How the Script Works

The script performs the following actions:

- Imports the necessary `um` module (and `sys` for exit codes).
- Defines constants for serial port parameters, the command to send (an AT command `ATI` which typically requests modem identification), and a timeout for waiting for a response.
- The `main()` function is the entry point.
- It calls `um.com_open()` to open and configure the specified serial port.

- If opening fails (indicated by a negative file descriptor `fd`), it prints an error and exits.
- If successful, it calls `um.com_xmit()` to send the `COMMAND_TO_SEND` and wait up to `RESPONSE_TIMEOUT_SEC` seconds for a response.
- It then prints the sent command and the received data (if any). The received data is decoded from bytes to a string, replacing any non-decodable characters.
- A `try...finally` block ensures that `um.com_close()` is called to close the serial port, even if an error occurs during communication.
- The standard `if __name__ == "__main__":` idiom ensures `main()` is called when the script is executed directly.

Step 2: Creating the Router App Package Files (Simplified)


This step involves preparing the necessary files for the ModulesSDK to package the Python script as a Router App. For this simple command-line script, we do not need complex `init`, `install`, or `uninstall` scripts. The SDK's default packaging will usually place the script in a `bin/` directory within the Router App's installation path (e.g., `/opt/<app_name>/bin/`). We will also omit metadata files like `name` and `version` for this basic example, though they are recommended for more complete Router Apps.

The primary file we need is our Python script itself (`serial_um_example.py`), which should be placed in the module's `source/` directory within the ModulesSDK.

Step 3: Preparing Makefiles for ModulesSDK Integration

The Advantech ModulesSDK uses a Makefile-based build system. To integrate our `serial_um_example` Python application:

1. **Create the module directory:** If it doesn't already exist, create a directory for your new module within the SDK. For example: `ModulesSDK/modules/serial_um_example/`.
2. **Place the Python script:** Copy or move your `serial_um_example.py` script into `ModulesSDK/modules/serial_um_example/source/serial_um_example.py`.
3. **Copy the main Makefile template:** Copy the generic module `Makefile` from `ModulesSDK/modules/template/Makefile` to your new module's directory: `ModulesSDK/modules/serial_um_example/Makefile`. This main Makefile generally handles the overall process of building the module for different platforms and creating the `.tgz` package. It usually calls the `Makefile` within the `source/` directory.
4. **Create the source Makefile:** Inside the `ModulesSDK/modules/serial_um_example/source/` directory, create a new `Makefile` with the following content. This Makefile defines how your Python script and any associated `um` module dependencies are installed into the Router App package.

Content for ModulesSDK/modules/serial_um_example/source/Makefile:


```
include ../../../../Rules.mk

all:
    @true

clean:
    @true

install:
    @install -d $(DESTDIR)/bin
    @install -m 644 $(SDKDIR)/library/$(OBJDIR)/*.so $(DESTDIR)/bin/
    @install -m 644 $(SDKDIR)/library/*.py $(DESTDIR)/bin/
    @install -m 755 *.py $(DESTDIR)/bin/
```

Resulting Directory Structure for the serial_um_example Module within SDK:

After these steps, the directory structure for your serial_um_example module within the ModulesSDK should look like this:

```
ModulesSDK/
|-- Rules.mk                      (and other root SDK files)
|-- library/                      (Example location for um.py and libum.so)
|   |-- um.py
|   |-- v4/                      (Platform-specific subdirectories)
|   |   `-- libum.so             (or libum.v4.so)
|   |-- v4i/
|   |   `-- libum.so
|   |-- ...
|-- modules/
|   |-- serial_um_example/
|   |   |-- source/
|   |   |   |-- serial_um_example.py (Your Python script)
|   |   |   `-- Makefile             (Source Makefile specific to this module)
|   |   `-- Makefile                 (Main module Makefile, copied from template)
|   |-- template/
|   |   `-- Makefile                 (Original template main Makefile)
|   |-- ...                          (Other example modules)
`-- ...                             (Other SDK directories)
```

Step 4: Building the Router App Package

Once the Python script and Makefiles are correctly placed within the ModulesSDK structure:

1. Navigate to the SDK's root directory:

```
user@machine:$ cd /path/to/your/ModulesSDK/
```

2. Build for all modules: Often, simply running `make` from the root of the SDK might build all modules for all default platforms if the main SDK Makefile is structured that way.

```
user@machine:/ModulesSDK$ make
```

The SDK's build system will invoke the Makefiles you prepared. The `install` target in your `source/Makefile` will copy `serial_um_example.py`, `um.py`, and the appropriate `libum.so` into the staging area. The main module Makefile will then package these files into a `*.tgz` archive.

Step 5: Uploading and Testing the Router App

Upload the generated `*.tgz` archive file for your target platform (e.g., `serial_um_example.v4.tgz`) to your Advantech router. This is typically done via the router's web interface, in the section *Customization* → *Router Apps*.

After the Router App is installed by the system:

- The Python script, along with `um.py` and `libum.so`, should be installed into a directory under `/opt/`, typically `/opt/serial_um_example/bin/`.
- **Test from the router's CLI:**
 - Run script from any location using the full path:
`/opt/serial_um_example/bin/serial_um_example.py`
 - Observe the output. The script will attempt to communicate with `/dev/ttyS0`. If a device is connected and responds to "ATI", you should see its response. Otherwise, you'll see a "No data received" message or an error if the port cannot be opened.

This example provides a basic framework for packaging a Python script that uses the `um` module as a Router App. For more complex applications, you might need to include more sophisticated `install`/`uninstall` scripts, manage dependencies, or handle background services with an `init` script.

12. USB Interface

12.1 Storage Access – USB Flash and SD Card

Connecting a USB device or SD card works in the standard Linux way. When you connect a USB Flash drive to the router, its device node will appear in the `/dev` directory. You can view details about detected devices using the `dmesg` command.

- **USB Flash drive** partitions typically appear as `/dev/sda1`. You can mount them using the `mount` command (e.g., `mount -t vfat /dev/sda1 /mnt`).
- Some **USB-to-Serial converters** are supported and will show up as `/dev/ttyUSB0`, `/dev/ttyUSB1`, etc. (See Section [12.4 Supported USB Serial Converter Chips](#)).
- An **SD Card** inserted into the router's reader usually appears with partitions like `/dev/mmcblk0p1`. You can mount it similarly (e.g., `mount -t vfat /dev/mmcblk0p1 /mnt`).

12.2 Mounting a USB Flash Drive Partition

To access files on a USB flash drive partition within the router's system, it must first be mounted. Follow these steps:

1. **Connect the USB Flash Drive:** Plug the USB flash drive into the router's USB port.
2. **Identify the Device Partition:** Run `dmesg | tail` to display recent system messages. Look for lines indicating the new device name (e.g., `sda`) and its partitions (e.g., `sda1`). Note the partition identifier, such as `/dev/sda1`.
3. **Create a Mount Point (Optional but Recommended):** Create an empty directory where the filesystem will be mounted. Using `/mnt` is common practice. `mkdir -p /mnt/usb`
4. **Mount the Partition:** Use the `mount` command to attach the partition to the mount point. The system often auto-detects the filesystem type. `mount /dev/sda1 /mnt/usb`
5. **Verify Successful Mount:** List mounted filesystems using `mount | grep /mnt/usb` or check the contents of the mount point directory (`ls /mnt/usb`) to confirm access.
6. **Unmount the Partition:** Before physically removing the drive, unmount it using the mount point or device name to prevent data corruption. `umount /mnt/usb` or `umount /dev/sda1`

Once unmounted, the USB flash drive can be safely removed. Ensure the correct device name and filesystem type (if specifying manually) are used.



If the mount command fails, double-check the device name (`/dev/sda1`) and try specifying the filesystem type with the `-t` option: `mount -t vfat /dev/sda1 /mnt/usb`.

12.3 Automount USB Flash Disk



This script provides a basic mechanism to automatically mount the first partition of a detected USB flash drive to `/mnt/flash` when inserted, and unmount it when removed. It requires firmware version 4.0.0 or later. The monitoring script should be saved to a file (e.g., `/root/automount.sh`) and launched via the Startup Script.

This example demonstrates how to create a background script that monitors for the presence of a USB flash drive and automatically mounts/unmounts its first partition.

Monitoring Script (`automount.sh`)

Save the following code into a file, for example, `/root/automount.sh` .



```
#!/bin/sh
#
LAST=0
i=0
while true
do
flsh=`cat /proc/diskstats |awk '/8\x20\x20\x20\x20\x20\x20\x201/ {print $3}'`
if [ $flsh ]; then
    i=1
else
    i=0
fi
if [ $LAST != $i ]; then
    LAST=$i
    if [ $i = 1 ]; then
        echo "Mount flash disk."
        if [ -d /mnt/flash ]; then
            mount /dev/$flsh /mnt/flash
        else
            mkdir /mnt/flash
            mount /dev/$flsh /mnt/flash
        fi
    else
        echo "UMOUNT flash disk."
        umount /mnt/flash
        rmdir /mnt/flash
    fi
fi
sleep 2
done
```


Startup Script

Add the following line to your Startup Script to launch the monitoring script in the background when the router boots. Ensure the path (`/root/automount.sh`) matches where you saved the monitoring script.

```
#!/bin/sh

# Launch the automount script in the background
sh /root/automount.sh &
```

How It Works

- The Startup Script simply executes the saved `automount.sh` script in the background using `sh ... &`.
- The `automount.sh` script runs an infinite loop (`while true`).
- Inside the loop, it reads `/proc/diskstats`, a kernel interface providing disk I/O statistics.
- It uses `awk` to search for a line matching the pattern `/ 8 1 /`. This pattern specifically looks for:
 - A space.
 - The major device number `8` (commonly used for SCSI/SATA/USB block devices like `/dev/sdX`).
 - Exactly seven spaces (`x20` represents a space in the original `awk` pattern).
 - The minor device number `1` (commonly representing the first partition, e.g., `sda1`).

If a matching line is found, `awk` prints the third field (`$3`), which is the device name (e.g., `sda1`).
- The device name is stored in the `flsh` variable.
- `if ["$flsh"]` : This checks if the `flsh` variable is non-empty. If the device partition was found, the variable will contain its name (like `sda1`), and the condition is true, setting `i` to 1 (detected). Otherwise, `i` is set to 0 (not detected). Quotes around `"$flsh"` prevent potential errors if the variable is empty.
- `if [$LAST != $i]` : The script compares the current detection state (`i`) with the state from the previous loop iteration (`LAST`). It only proceeds if the state has changed (device inserted or removed).
- `LAST=$i` : Updates the stored state for the next iteration.
- **If state changed to 1 (Device Detected):**
 - Prints "Mount flash disk."
 - Checks if the directory `/mnt/flash` exists using `[! -d ...]`. If it doesn't exist, it creates it using `mkdir`.
 - Executes `mount /dev/$flsh /mnt/flash` to mount the detected partition (e.g., `/dev/sda1`) onto the `/mnt/flash` directory. Filesystem type is typically auto-detected.
- **If state changed to 0 (Device Removed/Not Detected):**
 - Prints "UMOUNT flash disk."
 - Executes `umount /mnt/flash` to unmount the filesystem.

- Executes `rmdir /mnt/flash` to remove the mount point directory. Note that `rmdir` will fail if the directory is not empty (e.g., if the unmount failed or files were created outside the mount).
- `sleep 2` : The script pauses for 2 seconds before repeating the loop.



The method used to detect the USB drive by parsing `/proc/diskstats` for the specific pattern `/ 8 1 /` is very basic and potentially fragile. It will likely only work for the **first partition** (`sda1`) of the **first detected USB drive**. It may fail if the drive uses different major/minor numbers, has multiple partitions you wish to access, or if other block devices interfere. More robust solutions often involve using `udev` rules or dedicated automount daemons if available on the system.



After adding the `automount.sh` script (e.g., to `/root/`) and configuring the Startup Script to launch it, reboot the router. When you insert a compatible USB flash drive, its first partition should automatically become accessible under `/mnt/flash`.

12.4 Supported USB Serial Converter Chips

Advantech routers include built-in kernel support (drivers) for several common USB-to-serial converter chipsets. When an adapter using one of these chips is connected, the corresponding kernel module should load automatically, and the adapter should appear as a serial device node in the `/dev` directory (e.g., `/dev/ttyUSB0`).

Supported chip families generally include:

- **FTDI:** FT232R, FT232H, FT2232, FT4232, FT230X, etc. (Driver: `ftdi_sio`)
- **Silicon Labs:** CP210x series (e.g., CP2101, CP2102, CP2104) (Driver: `cp210x`)
- **Prolific:** PL2303 (various versions, support might vary) (Driver: `p12303`)
- **CDC-ACM:** Standard class for many modern USB serial devices (e.g., based on CH340/CH341 - support might depend on kernel config, some newer Arduino boards). (Driver: `cdc-acm`)

When you connect a supported adapter, the corresponding kernel modules load automatically, and the device appears as `/dev/ttyUSB0` , `/dev/ttyUSB1` , etc.

You can verify this in console with: `dmesg | grep -i ttyUSB`

12.5 Using an Unsupported Serial Converter Chip

In some cases, you may need to use a USB-to-Serial converter whose specific Vendor ID (VID) and Product ID (PID) combination is not natively recognized by the pre-loaded kernel drivers (like `ftdi_sio` or `p12303`), even if the underlying chip is technically supported by the driver. Every USB device is identified by these two hexadecimal numbers:

- **Vendor ID (VID):** Identifies the device manufacturer (e.g., `0403` for FTDI).
- **Product ID (PID):** Identifies the specific product model (e.g., `6001` for FT232R).

Finding the VID and PID

You can discover the VID and PID of your USB device in several ways:

- **On a Linux PC:** Use the `lsusb` command. The output lists connected devices with their IDs in `VID:PID` format. Note that `lsusb` command is not available on the router itself.
- **On Windows:** Open *Device Manager*, find the device (it might appear as an unknown device or under "Ports (COM & LPT)" or "Universal Serial Bus controllers"), right-click, select *Properties*, go to the *Details* tab, and choose "Hardware Ids" from the Property dropdown. Look for a string like `USB\VID_xxxx&PID_yyyy` .
- **On the Router:** Check kernel messages using `dmesg` immediately after plugging in the device. Look for lines like `usb 1-1: new full-speed USB device number X using ...` and potentially lines showing `idVendor=xxxx, idProduct=yyyy` .

Dynamically Enabling the Device via `sysfs`

Once you have the VID and PID (as four-digit hexadecimal numbers without the '0x' prefix) and know the appropriate kernel driver module name for the chip type (e.g., `ftdi_sio`, `pl2303`, `cp210x`), you can attempt to dynamically tell the driver to handle this specific VID/PID combination using the `new_id` interface within the `sysfs` filesystem:

Syntax: `echo <VID> <PID> > /sys/bus/usb-serial/drivers/ftdi_sio/new_id`

Replace `<VID>` and `<PID>` with the four-digit hex values (without `0x`).

Example (for VID=0403, PID=d921):

```
echo 0403 d921 > /sys/bus/usb-serial/drivers/ftdi_sio/new_id
```

After running this command, check `dmesg` again. If successful, you should see messages indicating the driver has claimed the device and created a serial port device node (e.g., `/dev/ttyUSB0`). If not, the driver might not support the underlying chip type.



This dynamic binding is temporary and will be lost on reboot. To make it persistent, this command must be executed from a Startup Script or a Router App's init script each time the router boots or the device is connected.

13. User LED

13.1 led Utility

The `led` utility provides basic command-line control over the user-controllable LED (often labeled "USR" or similar) typically found on the router's front panel.

Synopsis

```
led [on | off]
```

Option	Description
on	Turns the USR LED on (solid state).
off	Turns the USR LED off.

Table 7.: led Utility Options

13.2 Check IPsec Connection Status via LED



This script monitors the status of a specific IPsec tunnel and uses the USR LED to indicate whether the tunnel is established. It utilizes the `swanctl` command (part of strongSwan) and standard Linux utilities like `awk` and `grep`. The script should be saved to a file (e.g., `/root/ipsec_stat.sh`) and launched from the Startup Script.

This example provides a simple script to visually indicate the status of an IPsec connection using the router's USR LED. It checks the status every 5 seconds.

Monitoring Script (`ipsec_stat.sh`)

Save the following code into a file, for example, `/root/ipsec_stat.sh`. Adjust the `num` variable to match the IPsec connection number you want to monitor (typically 1, 2, 3, or 4, corresponding to the configuration order in the GUI).



```
#!/bin/sh
# ipsec_stat.sh - Monitors IPsec tunnel status and controls USR LED

num=1 # number of IPSec connection to monitor [1,2,3,4]

while true
do
    # List Security Associations and filter for the desired IPsec connection
    # then check if the line contains "INSTALLED"
    /usr/libexec/ipsec/swanctl --list-sas | awk "/ipsec$num/" | grep INSTALLED
    sts=$? # Capture the exit status of grep (0 if INSTALLED found, non-zero otherwise)

    # Control USR LED based on the status
    if [ "$sts" = "0" ]; then
        led on # Turn LED ON if tunnel SA is INSTALLED
    else
        led off # Turn LED OFF otherwise
    fi

    # Wait before next check
    sleep 5
done
```

Startup Script

Add the following line to your Startup Script (*Configuration* → *Scripts* → *Startup Script*) to launch the monitoring script in the background when the router boots. Ensure the path (`/root/ipsec_stat.sh`) is correct.

```
#!/bin/sh

# Launch the IPsec status monitoring script in the background
sh /root/ipsec_stat.sh &

exit 0
```

How It Works

- The Startup Script executes the saved `ipsec_stat.sh` script in the background using `sh ... &`.
- The `ipsec_stat.sh` script first sets a variable `num` to specify which IPsec connection instance (1-4) to monitor.
- It enters an infinite loop (`while true`).
- Inside the loop:
 - `/usr/libexec/ipsec/swanctl --list-sas` : This command lists the current IPsec Security Associations (SAs), providing detailed status information for active tunnels.
 - `| awk "/ipsec$num/"` : The output of `swanctl` is piped to `awk` . This filters the lines to only include those containing the specific connection name pattern (e.g., `ipsec1` if `num=1`).
 - `| grep INSTALLED` : The filtered output is then piped to `grep` to check if the line contains the word "INSTALLED". A successfully established SA typically shows this state.
 - `sts=$?` : The exit status of the `grep` command is captured in the variable `sts` . `grep` returns 0 if it finds a match ("INSTALLED" was found for the specified tunnel), and a non-zero value otherwise.
 - `if ["$sts" = "0"]` : The script checks if the exit status is 0.
 - * If `sts` is 0 (tunnel is installed/established), it executes `led on` to turn the USR LED on.
 - * If `sts` is non-zero (tunnel is not installed or the line wasn't found), it executes `led off` to turn the USR LED off.
 - `sleep 5` : The script pauses for 5 seconds before repeating the check.
- This provides a continuous visual indication of the specified IPsec tunnel's status via the USR LED.

After creating the `ipsec_stat.sh` script (e.g., in `/root/`), setting the correct IPsec connection number in the `num` variable, adding the launch command to the Startup Script, and rebooting the router, the USR LED should light up when the monitored IPsec tunnel is established.

13.3 Indicate OpenVPN Status via LED



Advantech routers lack an OpenVPN "management" port, making it difficult to precisely determine if the VPN connection is fully established. However, OpenVPN supports executing scripts when the tunnel process starts (`--up`) and stops (`--down`). This example uses these scripts to control the USR LED, providing a visual indication that the OpenVPN process is running, though **not necessarily that the connection is fully established and passing traffic**.

This example uses simple scripts triggered by OpenVPN's start and stop events to control the router's USR LED.

Up Script (`ledon.sh`)

Create a file, for example `/root/ledon.sh`, with the following content:



```
#!/bin/sh
# ledon.sh - Executed by OpenVPN on --up event
led on
```

Down Script (`ledoff.sh`)

Create another file, for example `/root/ledoff.sh`, with the following content:



```
#!/bin/sh
# ledoff.sh - Executed by OpenVPN on --down event
led off
```

Configuration

1. Create the two script files (`ledon.sh` and `ledoff.sh`) as shown above.
2. Make them executable: `chmod +x /root/ledon.sh /root/ledoff.sh`
3. Copy these script files to a persistent location on the router (e.g., `/root/`).
4. In the OpenVPN configuration settings within the router's web GUI (*Configuration -> VPN -> OpenVPN*), add the following line to the *Extra Options* field:

```
script-security 2
up /root/ledon.sh
down /root/ledoff.sh
```

Note: Enter each option on a new line in the Extra Options field.

How It Works

- The `script-security 2` option allows OpenVPN to call external scripts. It's crucial for security that this level is used, and the scripts themselves are secure.
- The `up /root/ledon.sh` option tells OpenVPN to execute the `ledon.sh` script after the tunnel device (e.g., `tun0`) has been successfully opened and configured. This script simply runs the `led on` command, turning the USR LED on.

- The `down /root/ledoff.sh` option tells OpenVPN to execute the `ledoff.sh` script when the OpenVPN tunnel process stops or the tunnel device is closed. This script runs `led off`, turning the USR LED off.
- This setup provides a basic visual cue: the LED is on when the OpenVPN process believes the tunnel is up, and off when it's stopped or down.



As noted, this method only indicates if the OpenVPN *process* has successfully executed its 'up' script. It does not guarantee that the VPN tunnel connection itself is successfully established, authenticated, or functional for passing traffic.

Part V.

Constraints

14. S1 Router Programming Considerations

The S1 Router line has specific characteristics affecting Router App development, primarily related to its security model and read-only filesystem.

14.1 Extending the Read-Only Root Filesystem

On S1 Routers, the root filesystem is mounted read-only and cannot be directly modified. However, functionality can be extended using Router Apps, which are installed as read-only SquashFS overlays mounted over specific directories like `/opt` and potentially `/usr`.

Other writable partitions, such as the persistent `/var` and volatile `/run` and `/tmp` directories, are mounted with the "noexec" option. This prevents the execution of binaries directly from these locations. Therefore, executable files (binaries, scripts intended for direct execution) must be packaged within a Router App and placed in appropriate locations within the overlay (e.g., `/opt/<RName>/bin`).

Examples `example8` and `example9` within the *ModulesSDK* demonstrate how to structure a Router App to add files to the `/usr` directory via the overlay mechanism. To build such an example (e.g., for the RBv2i-S1 platform):



```
cd modules/example8
make PLATFORM=RBv2i-S1
```

This build process generates a `*.raw` file. This `.raw` file is the Router App package for S1 platforms and can be installed using the standard Router App installation procedure in the web interface.

14.2 Adding JavaScript and CSS to the S1 Web Administration Interface

The S1 router's web server enforces a strict *Content-Security-Policy (CSP)*. Inline JavaScript code and CSS styles are generally blocked unless specific mechanisms are used. To include custom JavaScript or CSS within web pages generated by your Router App (e.g., CGI scripts) on S1 routers, they must be embedded within `<script>` or `<style>` tags that include a dynamically generated `nonce` (number used once) attribute.

The *ModulesSDK* (version 2.1.1 and later) provides helper functions, `um_html_js` and `um_html_css`, designed to handle this correctly for C-based CGI. These functions automatically incorporate the required nonce when embedding scripts or styles. Example usage within a C-based CGI script using the SDK:



```
// Example: Set focus on the username field after the page loads
um_html_js("document.f.username.focus();\n");

// Example: Add a simple CSS rule
um_html_css(".my-class { color: blue; }\n");
```


For Python CGI scripts on S1, ensure your HTML generation logic can incorporate a nonce if provided by the system or the `um` Python module for S1.

14.3 Changing System Configuration Programmatically on S1 Routers

Configuration files on S1 Router line devices include an integrity check (checksum) to prevent unauthorized modifications. To change system configuration settings from within a Router App script, you cannot simply edit the configuration files directly. Instead, follow this procedure:

1. Create a partial configuration backup file (e.g., `/tmp/backup.cfg`) containing only the settings you wish to change, formatted correctly (e.g., `SECTION_OPTION='value'`).
2. Calculate the SHA512 hash of this partial backup file and append it to the file itself in the format `INTEGRITY=CFG_HASH=<hash_value>`.
3. Use the `restore` command (with appropriate privileges, usually via `sudo` if available/required in the script context) to apply the changes from this integrity-checked partial backup file.

Example script snippet:



```
#!/bin/sh
# Create a temporary partial config file
cat << EOF > /tmp/backup.cfg
NETWORK_LAN_IPADDR='192.168.1.100'
NETWORK_LAN_NETMASK='255.255.255.0'
EOF

# Calculate hash and append integrity line
# Note: awk field might be $1 if output is just the hash
HASH_VAL=$(openssl sha512 /tmp/backup.cfg | awk '{print $2}')
echo "INTEGRITY=CFG_HASH=$HASH_VAL" >> /tmp/backup.cfg

# Apply the configuration change (requires appropriate permissions)
# The use of 'sudo' depends on the execution context of the script.
# If the script itself runs as root (e.g. some init), sudo might not be needed.
# Otherwise, ensure the user/context has sudo rights for 'restore'.
sudo restore /tmp/backup.cfg

# Clean up temporary file
rm /tmp/backup.cfg
```

Note: Ensure the user/context running this script has permissions to execute `sudo restore`. The exact mechanism may vary depending on how your script is invoked by the system.

15. Hardware Constraints

15.1 Non-volatile Memory

Routers are embedded systems with limited resources. Efficient use of storage, RAM, and CPU is crucial.

Router Apps are installed in the `/opt` directory. Additionally, persistent data that needs to survive reboots but is generated or modified at runtime can often be stored in the `/var/data` directory. The type and size of non-volatile memory available for these directories vary depending on the router platform, as shown in Table 8 and Table 9.

Parameter	v2i	v2i with eMMC	v3	v3 with eMMC	v4	v4i
Memory type	NOR	eMMC	MRAM	eMMC	eMMC	eMMC
File system	JFFS2	ext4	JFFS2	ext4	ext4	ext4
Part. size	2 MiB	512 MiB	128 KiB	512 MiB	512 MiB	474 MiB

Table 8.: Characteristics of the `/var/data` Directory Partition

Parameter	v2i	v2i with eMMC	v3	v3 with eMMC	v4	v4i
Memory type	NOR	eMMC	NOR	eMMC	eMMC	eMMC
File system	JFFS2	ext4	JFFS2	ext4	ext4	ext4
Part. size	12 MiB	814 MiB	128 MiB	838 MiB	838 MiB	2.16 GiB

Table 9.: Characteristics of the `/opt` Directory Partition



On platforms where the `/var/data` partition uses MRAM and is 128 KiB in size (e.g., standard v3 routers), it is strongly recommended that a Router App uses **no more than 64 KiB** of this space. The remaining space is required by the router's operating system for its own persistent data storage.

Notes:

- The JFFS2 filesystem used on some NOR flash partitions supports compression. This means you might be able to store more data than the raw partition size suggests if the data compresses well.
- It is standard practice for a Router App needing persistent runtime data storage to create its own subdirectory within `/var/data`, e.g., `/var/data/<RName>`. This should typically be done in the `install` or `init start` script.
- The system typically automatically deletes the `/var/data/<RName>` subdirectory (if it exists) when the corresponding Router App is uninstalled.
- Cleanup of any other files or subdirectories created by the Router App outside of `/opt/<RName>` or `/var/data/<RName>` is the responsibility of the Router App author (usually handled in the `uninstall` script).

15.2 RAM Utilization

Refer to Table 10 for the amount of RAM available on different router platforms. Router Apps can use standard dynamic memory allocation functions (e.g., `malloc` in C/C++, equivalent mechanisms in Python). Exercise caution regarding memory consumption; ensure your Router App does not deplete system memory, which could negatively impact router stability and performance.

Parameter	v2i routers	v3 routers	v4 routers	v4i routers
RAM size	128 MB	512 MB	1024 MB	1024 MB

Table 10.: RAM Memory Parameters

15.3 CPU Performance Considerations

There are CPU parameters for different router platforms listed in the Table 11. When developing applications consider this CPU constraints:

- Choose the development language appropriate for the task. C/C++ generally offers better performance for CPU-intensive operations than interpreted languages like Python or shell scripts.
- Avoid busy-waiting or tight loops that can consume excessive CPU.
- Offload complex computations to external systems if feasible and appropriate for your application.
- Profile your application if performance issues arise.

Parameter	v2i routers	v3 routers	v4 routers	v4i routers
CPU	SAM9X60	AM3352	ARMv8-A	ARMv8-A
Architecture	arm 5TEJ	arm v7	arm v8	arm v8
Core	ARM926EJ-S	Cortex-A8	Cortex-A72	Cortex-A53
CPU power	660 DMIPS	2000 DMIPS	4.7 DMIPS/MHz	2.3 DMIPS/MHz

Table 11.: CPU Architecture

Understanding Cross-Compilation and Compiler Flags

Cross-compilation is the process of compiling code on one system (the development host, e.g., x86 Linux) to run on a different system (the target router, e.g., ARM). This requires a toolchain (compiler, linker, libraries) built for the target architecture.

If you choose to use a different cross-compiler than the ones officially provided (see section 8.2), or if you need to fine-tune build parameters, ensure you use appropriate compiler flags for the target platform to generate compatible binaries. Based on the router platform, use the following flags:

v2i routers (ARMv5TEJ): `-march=armv5te -mtune=arm926ej-s -mfloat-abi=soft`

v3 routers (ARMv7-A / Cortex-A8): `-march=armv7-a -mtune=cortex-a8 -mfpv=vfpv3 -mfloat-abi=softfp`

v4 routers (ARMv8-A / Cortex-A72): `-march=armv8-a+crc+crypto -mtune=cortex-a72`

v4i routers (ARMv8-A / Cortex-A53): `-march=armv8-a+crc+crypto -mtune=cortex-a53`

Part VI.

Custom Firmware Compilation

16. Getting Started with Custom Firmware Compilation

16.1 Overview of Custom Firmware for Advantech Routers

Compiling custom firmware for Advantech routers involves modifying or rebuilding components of the router's operating system (ICR-OS). This can range from recompiling individual open-source utilities to, in theory, building a more comprehensive firmware image. This process is intended for advanced users who understand the implications and risks involved.

It is important to note that while it might be technically possible to build a complete firmware image using the provided sources, the router hardware will typically reject firmware images that are not digitally signed by Advantech. Therefore, the primary focus of custom compilation for end-users is often the modification or replacement of individual open-source components.

16.2 Prerequisites and Essential Knowledge

To embark on custom firmware compilation or modification of its components, certain prerequisites are essential:

- **Operating System:** A 64-bit Linux-based operating system is necessary for the build environment. This can be a dedicated physical machine or a virtual machine.
- **Technical Skills:** Familiarity with the Linux command line, basic C/C++ programming concepts (if modifying source code), and an understanding of build systems (like Make) are highly beneficial.
- **Hardware Access:** Access to the target Advantech router for testing and deployment is required.

Users are expected to have the necessary expertise to manage their Linux development environment.

16.3 Obtaining Firmware Source Code and Build System Components

All necessary development resources, including source code for open-source components and toolchains, are typically available through Advantech's [Bitbucket](#) repositories.

If your goal is to rebuild or modify an open-source component included in ICR-OS, you must first download the source code corresponding to your specific router firmware version. This can be found on the [Advantech ICR Source Code page](#). To determine your firmware version, navigate to the router's web interface, select *General* from the left menu, and check the *System Information* page. The firmware version follows a major.minor.patch format (e.g., 6.1.5). The corresponding source archive will typically be named similarly (e.g., `firmware-6.1.5-src.tar`).

The essential first step for any C/C++ development, including modifying firmware components, is to obtain and set up the appropriate cross-compiler for your target router platform. This enables you to generate executable binaries on your development machine that can run on the router. Details on acquiring and setting up the cross-compiler are provided in Chapter [8.3 SDK \(Software Development Kit\)](#) (within the Router Apps part of this manual).

17. Preparing the Build Environment

17.1 Setting Up the Development Host System

The instructions for setting up the build environment and its prerequisites have been validated on Ubuntu and Debian distributions. If you are using an RPM-based distribution (such as Fedora, which is also suitable), you will need to adapt the package installation commands to use your system's package manager (e.g., `dnf` or `yum`). Package names for dependencies might also differ slightly across distributions.

17.2 Setting Up the Cross-Compilation Toolchain

As mentioned in Section 16.3, a cross-compiler specific to your target Advantech router platform is crucial. This toolchain allows your development host (e.g., an x86-64 Linux machine) to compile code that will execute on the router's different processor architecture (typically ARM-based).

Please refer to Chapter 8.3 *SDK (Software Development Kit)* for detailed instructions on downloading, installing, and configuring the appropriate cross-compiler toolchain provided by Advantech. Ensure the toolchain's binaries are accessible in your system's `PATH` environment variable.

17.3 Configuring the Build System



This section would typically describe how to use tools like `menuconfig` or similar configuration interfaces if provided with the firmware source package to select components, features, and target platforms before starting the build process.

17.4 Understanding the Build Directory Structure



This section would describe the layout of the firmware source code directory, highlighting important sub-directories such as those containing package sources, kernel sources, toolchain integration, build scripts, and output locations for compiled binaries and images.

18. Building the Custom Firmware Image Components

18.1 The Firmware Build Process: Step-by-Step



Detailed build instructions for specific components are usually provided within the `README` file included in the downloaded source code archive for that component or the overall firmware source package. Generally, this involves navigating to the correct source directory and using `make` with appropriate targets.

18.2 Building and Integrating Open-Source Components

When rebuilding or modifying an open-source component from the ICR-OS sources:

- **Locate the Component:** Identify the specific component's source directory within the downloaded firmware source archive.
- **Consult README:** Detailed build instructions for individual components are typically provided within a `README` file in their respective source directories or in a general `README` for the entire source package.
- **Advantech Modifications:** Any modifications made by Advantech to the original open-source components are usually supplied as distinct patch files within the archive. This allows for easy identification and understanding of the changes applied by Advantech.
- **Compilation:** Follow the provided instructions to compile the component using the prepared cross-compilation toolchain. This usually involves standard commands like `make`.

The output will be the recompiled binary, library, or other files associated with that component.

18.3 Customizing the Linux Kernel (If Applicable)



Modifying the Linux kernel is a highly advanced task. If undertaken, it would involve obtaining the kernel source specific to the router, configuring it using kernel configuration tools (e.g., `make menuconfig` within the kernel source directory), compiling it with the cross-compiler, and then integrating the new kernel image. Replacing the kernel is extremely risky and can easily lead to an unbootable device.

18.4 Generating and Locating Firmware Image Files

When you recompile individual open-source components, the output is typically the specific binary or library file (e.g., an executable utility, a `.so` shared library). These files will be located in the build output directory specified by the component's build system, often within a subdirectory related to the target architecture.

As stated earlier, building a complete, digitally signable firmware image for direct flashing via standard update mechanisms is generally not feasible for end-users due to signature requirements. The process described here focuses on replacing individual, uncritical components.

18.5 Troubleshooting Common Build Issues

Common build issues include:



- Incorrectly configured cross-compiler path.
- Missing development libraries or tools on the host system (install them using your distribution's package manager).
- Incompatibility between the source code version and the toolchain.
- Typos in Makefiles or configuration scripts if modifications were made.

Always check the error messages carefully and consult the `README` files.

19. Installing and Managing Custom Firmware Components

19.1 Methods for Installing Built Firmware Components

After successfully building a component according to the instructions in the source archive's `README` file, the resulting binary or library files must be copied onto the target router's filesystem. This typically involves overwriting the original files. This process usually requires root access to the router, commonly achieved via SSH.

General Procedure (Non-S1 Routers):

1. Ensure you have root access to the router (e.g., via SSH).
2. Identify the location of the original component on the router's filesystem.
3. It is highly recommended to back up the original file before overwriting it.
4. Use a secure copy tool (like `scp`) to transfer the newly built file from your development machine to the appropriate location on the router, overwriting the original.
5. Verify file permissions and ownership of the newly copied file match the original.
6. Reboot the router or restart the relevant service for the changes to take effect (depending on the component).

S1 Router Line Devices: For S1 Router line devices, which feature a read-only root filesystem, installing custom components by directly overwriting files in system directories is generally not possible. Instead, such components often need to be packaged into a Router App that utilizes an overlay mechanism to present the modified files to the system. This is described in more detail in Chapter [14.1 Extending the Read-Only Root Filesystem](#).

19.2 Initial Boot-up and System Verification

After replacing a component and rebooting (if necessary):

- Check system logs for any new errors.
- Test the functionality related to the replaced component.
- If a utility was replaced, try running it from the command line with relevant options.



19.3 Important Note on Running Custom Firmware Components

Modifying or replacing standard firmware components with custom-built versions carries significant risks and consequences. Please consider the following points carefully:

- Even minor modifications to standard utilities or libraries can have unintended side effects, potentially disrupting system stability, breaking features, or causing router malfunctions.
- Advantech CZ R&D bears no responsibility for any issues, damages, or malfunctions resulting from the use of custom or modified firmware components. Such modifications may invalidate product certifications and guarantees.
- An Advantech router is a complex system comprising hardware, firmware (software), and certifications. Altering any of these layers results in a product that is fundamentally different from the one originally manufactured and certified by Advantech.
- Firmware components built solely from the provided open-source components might interact differently if they depend on Advantech's proprietary software which is not part of the open-source release. This means certain features may be affected, or full hardware support related to that component might be compromised.
- **Running custom or modified firmware components voids the device warranty if such modifications are found to be the cause of a malfunction.**
 - If modifications lead to non-compliance, all certifications listed on the device label and in the Declaration of Conformity could be considered invalid for that modified state. Telecommunication certification labels (e.g., PTCRB, GCF, RCM, FCC/IC) are tied to the specific certified firmware and hardware configuration.
 - Improper hardware handling resulting from firmware component modifications (e.g., incorrect power sequencing for peripherals due to a modified driver or utility) can shorten the device's lifespan or cause permanent hardware damage.
 - If you intend to distribute devices running your custom firmware components, and these significantly alter the product's certified behavior, the Advantech labels may no longer fully represent the product. Please contact Advantech business representatives to discuss options if extensive modifications are planned for commercial redistribution.
- Custom firmware components are not supported by Advantech Czech s.r.o. If a device becomes unbootable or unusable due to such modifications (e.g., corrupted critical library, damaged MRAM data due to a faulty custom utility), Advantech support will not cover repairs under warranty. Attempting repairs will incur service fees.
- Uploading custom firmware images (if a full image build was attempted and is possible to flash) to devices running standard ICR-OS (v3 generation and later) will trigger a warning about the missing digital signature. Proceeding past this warning is done at your own risk and confirms acceptance of the warranty voidance implications. (Note: Digital signature verification for full firmware images applies primarily to v3 and later generation devices.)

Exercise extreme caution when replacing system components: replacing critical system components, particularly the Linux kernel or essential libraries like libc, can potentially render the device unbootable ("bricked"), often without a straightforward recovery method. Modifying firmware components is done entirely at your own risk.

19.4 Post-Installation Troubleshooting

If issues arise after installing a custom component:

- **Revert Changes:** If possible, restore the original backed-up component.
- **Check Logs:** Examine system logs (*Status* → *System Log* in the web interface, or via `dmesg/logread` on the console) for error messages related to the modified component.
- **Serial Console:** Access via a serial console can provide boot messages and diagnostic information even if the network or SSH is unavailable.
- **Safe Mode/Recovery:** Familiarize yourself with any safe mode or firmware recovery procedures applicable to your router model (if available) **before** making critical changes.



Part VII.

Special Router Configuration Options

20. Overview of Special Configuration Options

20.1 Special Options List

This chapter describes advanced configuration parameters that cannot be configured via the web GUI. These options must be set either via SSH or by editing a startup script, which can be configured through the GUI.

Item	Description
PPP_TOLERATE_NO_SIM	When set to 1, module restarts are disabled if connectivity is enabled but SIM 1 is missing. After inserting SIM 1, the router must be rebooted manually.
PPP_TOLERATE_NO_SIM2	Same as above, but for SIM 2 (if available).
PPP_TOLERATE_NO_SIGNAL	When set to 1, module restarts are disabled when the signal is lost. Note that SIM switching will not function in this mode.
PPP_TOLERATE_NO_SIGNAL2	Same as above, but for SIM 2 (if available).


Table 12.: Summary of Special Router Configuration Options

20.2 How to Apply Special Options

To apply these special configuration options, you must directly edit the appropriate configuration file stored on the router. These files are located in the `/etc` directory and are named using the pattern `settings.*`. You need to locate the file that contains the desired parameter, which can usually be inferred from the parameter name. For example, a configuration parameter beginning with `PPP_` will be found in the `settings.ppp` file. See an example script on the next page.

20.3 Configuration by a Script

The following example shows how to set a special option using a shell script. The script below sets `PPP_REGISTRATION_TOUT` to 5 minutes:



```
#!/bin/sh

# Parameters
CONFIG_FILE="/etc/settings.ppp"
PARAM_NAME="PPP_REGISTRATION_TOUT"
PARAM_VALUE="300"

# Check if the file exists
if [ ! -f "$CONFIG_FILE" ]; then
    echo "File $CONFIG_FILE does not exist!"
    exit 1
fi

# Check if the parameter exists in the file
if grep -q "^${PARAM_NAME}=" "$CONFIG_FILE"; then
    # Parameter exists, update its value
    sed -i "s/^${PARAM_NAME}=.*/${PARAM_NAME}=${PARAM_VALUE}/" "$CONFIG_FILE"
    echo "Updated $PARAM_NAME to $PARAM_VALUE"
else
    # Parameter doesn't exist, add it to the end of the file
    echo "${PARAM_NAME}=${PARAM_VALUE}" >> "$CONFIG_FILE"
    echo "Added $PARAM_NAME=$PARAM_VALUE to the file"
fi

# Verify the change
if grep -q "^${PARAM_NAME}=${PARAM_VALUE}$" "$CONFIG_FILE"; then
    echo "Verification successful: $PARAM_NAME is now set to $PARAM_VALUE"
else
    echo "Verification failed: $PARAM_NAME was not properly set to $PARAM_VALUE"
    exit 1
fi
```

Part VIII.

Related Documents

Related Documents

You can obtain product-related documents on the **Engineering Portal** at icr.advantech.com.

To access your router's documents or firmware, go to the [Router Models](#) page, locate the required model, and select the appropriate tab below.

Documents that are common to all models and describe specific functionality areas are available on the [Application Notes](#) page.

The **Router Apps** installation packages and manuals are available on the [Router Apps](#) page.

If you are interested in further options for extending router functionality, either through scripts or custom Router Apps, please see the information available on the [Development](#) page.