Add-on アプリケーション開発
- パッケージ作成マニュアル -

リリース *1.2*

サン電子株式会社 M2M 事業部 ソフトウェア開発部

2021 年 01 月 12 日

目次

第 1 章

はじめに

この文書では NSX7000 用 Add-on アプリケーション・パッケージを作成するための手順について説明します.

下記の NSX7000 を対象とします.

- ファームウェア・バージョン 1.3.1 以降がインストールされている
- 開発者モードが ON である

1.1 取り扱う話題

この文書では下記の話題を取り扱います.

- Java で開発したアプリケーションを Add-on アプリケーション・パッケージにする手順
- C/C++ で開発したアプリケーションを Add-on アプリケーション・パッケージにする手順
- アプリケーションをデーモンにする方法
- ケーパビリティを指定してアプリケーションを実行する方法

1.2 対象読者

この文書は下記の知識・経験を有した読者を対象とします.

- 一般的な Linux ディストリビューション[*1] での基本的なコマンドライン操作に習熟している
- シェル・スクリプトに習熟している
- GNU make, および, その Makefile に習熟している

[*1] 例. Debian GNU/Linux, Ubuntu, Red Hat Linux など

- (Java で開発する場合) javac コマンドを使って Java アプリケーションをビルドできる
- (C/C++ で開発する場合) gcc/g++ コマンドを使って C/C++ アプリケーションをビルドできる

1.3 必要なもの

NSX7000 用 Add-on アプリケーションを開発するには一般的な Linux ディストリビューションがインストールされたホスト PC, もしくは仮想マシンが必要です.

上記の環境に SDK がインストールされている必要があります.

詳細については別文書「Add-on アプリケーション開発-環境構築マニュアル-」を参照してください.

1.4 NSX7000 が提供する実行環境

NSX7000 が提供する実行環境はつぎの表のとおりです.

表 1　NSX7000 が提供する実行環境

名称	実装	バージョン
シェル	BusyBox ash	1.24.2
Java ランタイム	AdoptOpenJDK	11.0.1+13.1[*2]
C ライブラリ	GNU C library	2.22
C++ 標準ライブラリ	GNU C++ Standard C++ library	6.0.21

1.5 SDK が提供する開発環境

SDK が提供する開発環境はつぎの表のとおりです.

表 2　SDK が提供する開発環境

名称	実装	バージョン
C コンパイラ	GNU C compiler	5.3.0
C++ コンパイラ	GNU C++ compiler	5.3.0
CMake	CMake	3.6.1

SDK は Java 開発環境を提供しません. 本文書にそって開発を行なうには,JDK(Java Development Kit) をインストールしてください.

[*2] インストール済み Java ランタイムのバージョンは NSX7000 の製造時期によって異なります.

1.5.1 Debian GNU/Linux 8 に JDK をインストールする

Debian GNU/Linux 8 での JDK のパッケージ名は「openjdk-7-jdk」です.

このパッケージをインストールするにはつぎのコマンドを実行します.

```
$ sudo apt install openjdk-7-jdk
```

1.5.2 Debian GNU/Linux 9, Ubuntu 16.04 LTS に JDK をインストールする

Debian GNU/Linux 9, および, Ubuntu 16.04 LTS での JDK のパッケージ名は「openjdk-8-jdk」です.

このパッケージをインストールするにはつぎのコマンドを実行します.

```
$ sudo apt install openjdk-8-jdk
```

第 2 章

Java アプリケーションをパッケージ化する

本章では Java アプリケーションをパッケージ化する方法について説明します.

まず, HelloWorld アプリケーションを作成し, 手動で NSX 上で実行する方法について説明します. その後, その HelloWorld アプリケーションをパッケージ化します.

つぎに, ネットワーク到達性を確認するアプリケーションを作成して, パッケージ化します. このアプリケーションをデーモンにする方法について説明します.

2.1 Java アプリケーションを手動で実行する

NSX7000 で Java アプリケーションを手動で実行する手順は下記のとおりです.

1. NSX7000 に「/app/package/**アプリケーション名**」ディレクトリを作成する
2. scp コマンドで Java アプリケーションと関連ファイルを「/app/package/**アプリケーション名**」ディレクトリ以下にコピーする
3. NSX7000 に SSH ログインし, /app/package/java-runtime/bin/java コマンド使って Java アプリケーションを実行する

以下では NSX7000 で HelloWorld アプリケーションを実行する例を示します.

2.1.1 例: NSX7000 で HelloWorld アプリケーションを実行する

HelloWorld.class クラスファイルを準備する

1. HelloWorld アプリケーションのソースコード HelloWorld.java を作成する

リスト 1　HelloWorld.java

```
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

2. javac コマンドで HelloWorld.java コンパイルし, HelloWorld.class クラスファイルを作成する

```
$ javac HelloWorld.java
```

NSX7000 にアプリケーション用ディレクトリを準備する

1. **suncorp** アカウントで NSX7000(IP アドレス: 192.168.62.1) に SSH ログインする

```
$ ssh suncorp@192.168.62.1
```

2. **su** コマンドを使って **root** アカウントに切り替える

```
suncorp@NSX:~$ su
Password:


BusyBox v1.24.2 () built-in shell (ash)

root@NSX:/home/suncorp#
```

3. HelloWorld アプリケーション用ディレクトリ/app/package/hello-world を作成する

```
root@NSX:/home/suncorp# mkdir /app/package/hello-world
```

4. **suncorp** アカウントで書き込めるように作成したディレクトリのグループを変更する

```
root@NSX:/home/suncorp# chgrp suncorp /app/package/hello-world
```

5. NSX7000 からログアウトする

```
root@NSX:/home/suncorp# exit
suncorp@NSX:~$ exit
Connection to 192.168.62.1 closed.
```

HelloWorld.class クラスファイルを配置する

1. scp コマンドを使って NSX7000 の/app/package/hello-world ディレクトリに HelloWorld.class クラスファイルをコピーする

```
$ target_dir=/app/package/hello-world/
$ scp HelloWorld.class suncorp@192.168.62.1:$target_dir
HelloWorld.class                              100%  425     0.4KB/s   00:00
```

HelloWorld アプリケーションを実行する

1. **suncorp** アカウントで NSX7000(IP アドレス: 192.168.62.1) に SSH ログインする

```
$ ssh suncorp@192.168.62.1
```

2. /app/package/java-runtime/bin/java コマンドを使って HelloWorld アプリケーションを実行する

```
suncorp@NSX:~$ java_runtime=/app/package/java-runtime/bin/java
suncorp@NSX:~$ app_dir=/app/package/hello-world
suncorp@NSX:~$ $java_runtime -classpath $app_dir HelloWorld
Hello World
```

2.2 Java アプリケーションをパッケージ化する

Java アプリケーションをパッケージ化する手順は下記のとおりです.

1. 規定のディレクトリと control ファイルを作成する
2. appctl スクリプトを作成する
3. パッケージ作成用 Makefile を作成する
4. パッケージを作成する

2.2.1 規定のディレクトリと control ファイルを作成する

Java アプリケーションをパッケージ化するには下記のディレクトリ構成と control ファイル, appctl スクリプト, パッケージ作成用 Makefile が必要です.

```
トップ・ディレクトリ/
|
+-- rpk/
|      |
|      +-- CONTROL/
|      |       |
|      |       +-- control ファイル
|      |
|      +-- appctl スクリプト
|
+-- Makefile
```

トップ・ディレクトリ名は任意です. 以下ではトップ・ディレクトリ名を hello-world として説明します.

1. hello-world トップ・ディレクトリ, rpk サブディレクトリ, CONTROL サブディレクトリを作成します.

```
$ mkdir -p hello-world/rpk/CONTROL
```

2. hello-world/rpk/CONTROL サブディレクトリに下記の control ファイルを作成します.

リスト 2　control ファイル

```
Package: @ADD_ON_PKG_NAME@
Version: @ADD_ON_PKG_VERSION@
Depends: @ADD_ON_PKG_DEPENDS@
Runtime-Depends: @ADD_ON_PKG_RUNTIME_DEPENDS@
Maintainer: @ADD_ON_PKG_MAINTAINER@
Architecture: @ADD_ON_PKG_ARCHITECTURE@
Provides: @ADD_ON_PKG_PROVIDES@
```

(次のページに続く)

(前のページからの続き)

```
Replaces: @ADD_ON_PKG_REPLACES@
Description: @ADD_ON_PKG_DESCRIPTION@
```

2.2.2 appctl スクリプトを作成する

appctl スクリプトを作成します.

appctl スクリプトとは下記の仕様に準拠するシェル・スクリプトです.

- 「start」サブコマンドを理解する
- 「stop」サブコマンドを理解する
- 「restart」サブコマンドを理解する

システムの起動時に **start** サブコマンドが指定されて appctl スクリプトが実行されます. start サブコマンド指定時にアプリケーションを起動する処理を実装します.

システムの終了時に **stop** サブコマンドが指定されて appctl スクリプトが実行されます. stop サブコマンド指定時にアプリケーションを終了する処理を実装します.

アプリケーションを再起動する際に **restart** サブコマンドが指定されて appctl スクリプトが実行されます. restart サブコマンド指定時にアプリケーションを再起動する処理を実装します.

ここでは下記の appctl スクリプトを用意します.

リスト 3　appctl スクリプト

```
#!/bin/sh

PACKAGE_NAME=@ADD_ON_PKG_NAME@
PACKAGE_DIR=/app/package

JAVA=${PACKAGE_DIR}/java-runtime/bin/java
CLASSPATH=${PACKAGE_DIR}/${PACKAGE_NAME}
CLASS=HelloWorld
OUTPUT_FILE=/app/var/hello-world-java.txt

start_app() {
  $JAVA -classpath $CLASSPATH $CLASS >$OUTPUT_FILE
}

stop_app() {
  rm -f $OUTPUT_FILE
}

case "$1" in
```

(次のページに続く)

(前のページからの続き)

```
  start)
    start_app
    ;;
  stop)
    stop_app
    ;;
  restart)
    stop_app
    start_app
    ;;
   *)
    ;;
esac

exit 0
```

標準出力への出力は/app/var/hello-world-java.txt にリダイレクトしています.

2.2.3 パッケージ作成用 Makefile を作成する

パッケージ作成用 Makefile を作成します.

ここでは下記の Makefile を用意します.

リスト 4　パッケージ作成用 Makefile

```
ROOSTER_TOP_DIR ?= $(HOME)/RoosterOS-SDK

ADD_ON_PKG_NAME := hello-world-java
ADD_ON_PKG_VERSION := 1.0
ADD_ON_PKG_DEPENDS := java-runtime
ADD_ON_PKG_RUNTIME_DEPENDS := java-runtime
ADD_ON_PKG_MAINTAINER := your-name@example.com
ADD_ON_PKG_DESCRIPTION := hello world application

include $(ROOSTER_TOP_DIR)/mk/add-on-package.mk

HelloWorld.class: HelloWorld.java
        javac $<

contents: $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) HelloWorld.class
        cp HelloWorld.class $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)
        touch $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED)


$(eval $(DefaultTarget))
```

ADD_ON_PKG_NAME　パッケージ名を記述します. ここでは hello-world-java としています.

ADD_ON_PKG_VERSION　パッケージのバージョンを記述します. ここでは 1.0 としています.

ADD_ON_PKG_DEPENDS　パッケージが依存している別のパッケージを記述します. Java アプリケーションは必ず **java-runtime** と記述してください.

ADD_ON_PKG_RUNTIME_DEPENDS　実行時にパッケージが依存している別のパッケージを記述します. Java アプリケーションは必ず **java-runtime** と記述してください.

ADD_ON_PKG_MAINTAINER　開発者のメールアドレスを記述します. ここでは your-name@example.com としています.

ADD_ON_PKG_DESCRIPTION　パッケージの説明文を記述します. ここでは「hello world application」としています.

include ディレクティブで SDK の add-on-package.mk を取り込みます. add-on-package.mk を取り込むことで後述の **rpk** ターゲットを利用できるようになります.

HelloWorld.class ターゲットで HelloWorld.java ソースコードをコンパイルします.

HelloWorld.java ソースコードが必要です. トップディレクトリに HelloWorld.java ソースコードを配置してください. HelloWorld.java を含めたディレクトリ構成は下記です.

トップ・ディレクトリ/
```
|
+-- rpk/
|      |
|      +-- CONTROL/
|      |       |
|      |       +-- control ファイル
|      |
|      +-- appctl スクリプト
|
+-- Makefile
|
+-- HelloWorld.java
```

contents ターゲットで HelloWorld.class クラスファイルを$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) ディレクトリにコピーし, $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED) ファイルを作成します. $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) ディレクトリにコピーしたものがパッケージの内容物になります.

2.2.4 パッケージを作成する

make rpk コマンドを実行してパッケージを作成します.

```
$ make rpk
```

成功すると **hello-world-java_1.0.rpk** ファイルが作成されます.

下記の手順でインストールしてください.

1. scp コマンドを使って NSX7000 の/tmp ディレクトリに hello-world-java_1.0.rpk をコピーする

```
$ scp hello-world-java_1.0.rpk suncorp@192.168.62.1:/tmp
hello-world-java_1.0.rpk                    100% 1662     1.6KB/s   00:00
```

2. suncorp アカウントで NSX7000(IP アドレス: 192.168.62.1) に SSH ログインする

```
$ ssh suncorp@192.168.62.1
```

3. su マンドを使って root アカウントに切り替える

```
suncorp@NSX:~$ su
Password:


BusyBox v1.24.2 () built-in shell (ash)

root@NSX:/home/suncorp#
```

4. **rpkg install** コマンドを実行して hello-world-java_1.0.rpk をインストールする

```
root@NSX:/home/suncorp# rpkg install /tmp/hello-world-java_1.0.rpk
UBI device number 4, total 768 LEBs (97517568 bytes, 93.0 MiB),
available 0 LEBs (0 bytes), LEB size 126976 bytes (124.0 KiB)
Installing hello-world-java (1.0) on root
Configuring hello-world-java.
Installing hello-world-java (1.0) on root
Configuring hello-world-java.
```

5. **/etc/init.d/rooster_os_application restart** コマンドを実行してアプリケーションを再起動する

```
root@NSX:/home/suncorp# /etc/init.d/rooster_os_application restart
app.img: OK
app.img: OK
```

実行後に下記を確認してください.

- /app/package/hello-world-java ディレクトリが作成されている
- /app/package/hello-world-java ディレクトリに HelloWorld.class クラスファイルがある
- /app/package/hello-world-java ディレクトリに sbin/appctl スクリプトがある

- /app/var/hello-world-java.txt ファイルが作成されている
- /app/var/hello-world-java.txt ファイルに「Hello World」と書き込まれている

アプリケーションを停止したときに何が起きるかを確認するため**/etc/init.d/rooster_os_application stop** コマンドを実行してください.

```
root@NSX:/home/suncorp# /etc/init.d/rooster_os_application stop
```

実行後に下記を確認してください.

- /app/package/hello-world-java ディレクトリが削除されている
- /app/var/hello-world-java.txt ファイルが削除されている

再度アプリケーションを開始するには**/etc/init.d/rooster_os_application start** コマンドを実行してください.

```
root@NSX:/home/suncorp# /etc/init.d/rooster_os_application start
app.img: OK
app.img: OK
```

2.3 Java アプリケーションをデーモン化する

Java アプリケーションをデーモン化する手順は下記のとおりです.

- パッケージ名と同名のスクリプトを作成する
- appctl からパッケージ名と同名のスクリプトを実行する
- appctl から実行されるスクリプトで, Java アプリケーションを Procd system manager に登録する処理を実装する

本節では例としてネットワーク到達性を定期的に確認する Java アプリケーションを作成します. この Java アプリケーションを Procd system manager に登録するスクリプトを実装し, Java アプリケーションをデーモン化します.

2.3.1 例: ネットワーク到達性を定期的に確認する Java アプリケーションをデーモン化する

ネットワーク到達性を定期的に確認する Java アプリケーションを作成します. パッケージ名は network-reachability-test-java とし, 本パッケージの仕様を下記とします.

- ネットワーク到達性を確認する送信先ホストはプロパティファイルで指定する
  - キーは *target_ipv4_address* とする
  - 値は IPv4 アドレスで指定する
- ネットワーク到達性を確認する ICMP echo リクエストの送信間隔はプロパティファイルで指定する
  - キーは *interval* とする
  - 値は非負の数値（単位は秒）で指定する
- ネットワーク到達性の確認には java.net.InetAddress クラスの isReachable メソッドを使用する
  - そのタイムアウト値はプロパティファイルで指定する
    - キーは *timeout* とする
    - 値は非負の数値（単位は秒）で指定する
- 第 1 引数にプロパティファイルのパスを指定する
- 第 2 引数にログファイルのパスを指定する

上記にもとづいて作成した Java のソースコードが NetworkReachabilityTest.java です.

リスト 5　NetworkReachabilityTest.java

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.InputStream;
import java.io.IOException;
import java.io.PrintStream;
import java.net.Inet4Address;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.Date;
import java.util.Properties;

public class NetworkReachabilityTest {
    // プロパティ名の定義
    private static final String PROP_TARGET_IPV4_ADDRESS = "target_ipv4_address";
    private static final String PROP_INTERVAL = "interval";
    private static final String PROP_TIMEOUT = "timeout";

    // プロパティのデフォルト値
    private static final String DEFAULT_TARGET_IPV4_ADDRESS = "127.0.0.1";
    private static final int DEFAULT_INTERVAL = 10;
    private static final int DEFAULT_TIMEOUT = 10;

    // ログファイルへの出力用ストリーム
    private static PrintStream _logFileStream = null;

    // 指定された path をログファイルとして扱う出力用ストリームを開く
    private static PrintStream openLogFileStream(String path)
        throws FileNotFoundException {
        File file = new File(path);
        return new PrintStream(file);
    }

    // ログファイルにログを書き込む
    private static void writeLog(String fmt, Object... args) {
        if (_logFileStream == null) return;

        Date date = new Date();
        _logFileStream.printf("%tF %tT: ", date, date);
        _logFileStream.printf(fmt, args);
        _logFileStream.flush();
    }

    // 指定された path をプロパティファイルとして読み込む
    private static Properties loadProperties(String path) throws IOException {
        InputStream is = new FileInputStream(path);
```

(次のページに続く)

(前のページからの続き)

```
        Properties prop = new Properties();
        try {
            prop.load(is);
        } finally {
            is.close();
        }

        return prop;
    }

    // デフォルトの Inet4Address を作成する
    private static Inet4Address getDefaultIPv4Address(String ipv4AddrStr) {
        try {
            return (Inet4Address)InetAddress.getByName(ipv4AddrStr);
        } catch (UnknownHostException e) {
            throw new RuntimeException("invalid default IPv4 address: "
                                       + ipv4AddrStr);
        }
    }

    // 指定された prop からキー name に対応する Inet4Address を取得する.
    // 取得できなければ defaultValue から Inet4Address を作成する
    private static Inet4Address getIPv4Address(Properties prop,
                                               String name,
                                               String defaultValue) {
        String value = prop.getProperty(name);
        if (value == null) {
            writeLog("property '%s' is not found%n"
                     + "use default value: '%s'%n",
                     name, defaultValue);
            return getDefaultIPv4Address(defaultValue);
        }

        InetAddress addr = null;
        try {
            addr = InetAddress.getByName(value);
        } catch (UnknownHostException e) {
            writeLog("property '%s' is not an IPv4 address: '%s'%n"
                     + "use default value: '%s'%n",
                     name, value, defaultValue);
            return getDefaultIPv4Address(defaultValue);
        }

        Inet4Address ipv4Addr = (Inet4Address)addr;
        if (ipv4Addr == null) {
            writeLog("property '%s' is not an IPv4 address: '%s'%n"
                     + "use default value: '%s'%n",
                     name, value, defaultValue);
```

(次のページに続く)

(前のページからの続き)

```
            return getDefaultIPv4Address(defaultValue);
        }

        return ipv4Addr;
    }

    // プロパティファイルから"target_ipv4_address"キーの値を Inet4Address として取得する
    private static Inet4Address getTargetIPv4Address(Properties prop) {
        return getIPv4Address(prop,
                              PROP_TARGET_IPV4_ADDRESS,
                              DEFAULT_TARGET_IPV4_ADDRESS);
    }

    // 指定された prop からキー name に対応する int を取得する.
    // 取得できない, もしくはその値が 0 以下なら defaultValue から int を作成する
    private static int getPositiveIntValue(Properties prop,
                                           String name, int defaultValue) {
        String str = prop.getProperty(name);
        if (str == null) {
            writeLog("property '%s' is not found%n"
                     + "use default value: '%d'%n",
                     name, defaultValue);
            return defaultValue;
        }

        int value = -1;
        try {
            value = Integer.parseInt(str);
        } catch (NumberFormatException e) {
            writeLog("property '%s' is not an integer value: '%s'%n"
                     + "use default value: '%d'%n",
                     name, str, defaultValue);
            return defaultValue;
        }

        if (value <= 0) {
            writeLog("property '%s' is not a positive integer value: '%i'%n"
                     + "use default value: '%d'%n",
                     name, value, defaultValue);
            return defaultValue;
        }

        return value;
    }

    // プロパティファイルから"interval"キーの値を int として取得する
    private static int getInterval(Properties prop) {
        return getPositiveIntValue(prop,
```

(次のページに続く)

(前のページからの続き)

```
                                    PROP_INTERVAL, DEFAULT_INTERVAL) * 1000;
    }

    // プロパティファイルから"timeout"キーの値を int として取得する
    private static int getTimeout(Properties prop) {
        return getPositiveIntValue(prop,
                                    PROP_TIMEOUT, DEFAULT_TIMEOUT) * 1000;
    }

    public static void main(String[] args) {
        if (args.length < 2) {
            System.err.println("need properties file path and log file path");
            System.exit(1);
        }

        String propertiesPath = args[0]; // プロパティファイルへのパス
        String logFilePath = args[1];    // ログファイルへのパス

        // ログファイルを開く
        try {
            _logFileStream = openLogFileStream(logFilePath);
        } catch (IOException e) {
            System.err.println("failed to open " + logFilePath);
            System.exit(1);
        }

        // アプリケーションの終了時にログファイルを閉じる
        Runtime.getRuntime().addShutdownHook(new Thread() {
                @Override
                public void run() {
                    _logFileStream.close();
                }
            });

        // プロパティファイルを読み込む
        Properties prop = null;
        try {
            prop = loadProperties(propertiesPath);
        } catch (IOException e) {
            System.err.println("failed to open " + propertiesPath);
            System.exit(1);
        }

        // プロパティファイルからターゲットの IPv4 アドレス, 送信間隔,
        // タイムアウトの設定値を取得する
        Inet4Address targetIPv4Addr = getTargetIPv4Address(prop);
        String targetIPv4AddrString = targetIPv4Addr.getHostAddress();
        int interval = getInterval(prop);
```

(次のページに続く)

(前のページからの続き)

```
        int timeout = getTimeout(prop);

        // 送信間隔ごとにネットワーク到達性を確認する
        try {
            while (true) {
                boolean ok = false;
                try {
                    ok = targetIPv4Addr.isReachable(timeout);
                } catch (IOException e) {
                    writeLog("network error occurred: " + e);
                    Thread.sleep(interval);
                    continue;
                }
                writeLog("'%s' is %s%n",
                         targetIPv4AddrString,
                         ok ? "reachable" : "unreachable");
                Thread.sleep(interval);
            }
        } catch (InterruptedException e) {}

        System.exit(0);
    }
}
```

プロパティファイル network-reachability-test-java.properties は下記です.

リスト 6　network-reachability-test-java.properties

```
target_ipv4_address = 192.168.62.100
interval = 5
timeout = 30
```

control ファイルは下記です. hello-world-java パッケージと同一です.

リスト 7　control ファイル

```
Package: @ADD_ON_PKG_NAME@
Version: @ADD_ON_PKG_VERSION@
Depends: @ADD_ON_PKG_DEPENDS@
Runtime-Depends: @ADD_ON_PKG_RUNTIME_DEPENDS@
Maintainer: @ADD_ON_PKG_MAINTAINER@
Architecture: @ADD_ON_PKG_ARCHITECTURE@
Provides: @ADD_ON_PKG_PROVIDES@
Replaces: @ADD_ON_PKG_REPLACES@
Description: @ADD_ON_PKG_DESCRIPTION@
```

appctl スクリプトは下記です.

リスト 8　appctl スクリプト

```
#!/bin/sh

PACKAGE_NAME=@ADD_ON_PKG_NAME@
PACKAGE_DIR=/app/package

exec $PACKAGE_DIR/$PACKAGE_NAME/sbin/$PACKAGE_NAME $@
```

appctl スクリプトから実行される network-reachability-test-java スクリプトは下記です.

リスト 9　network-reachability-test-java スクリプト

```
#!/bin/sh /etc/rc.common

USE_PROCD=1

PACKAGE_NAME=@ADD_ON_PKG_NAME@
PACKAGE_DIR=/app/package

JAVA=${PACKAGE_DIR}/java-runtime/bin/java
CLASSPATH=${PACKAGE_DIR}/${PACKAGE_NAME}
CLASS=NetworkReachabilityTest

PROPERTIES_FILE=${PACKAGE_DIR}/${PACKAGE_NAME}/${PACKAGE_NAME}.properties
LOG_FILE=/app/var/${PACKAGE_NAME}.log

start_service() {
  procd_open_instance
  procd_set_param command \
                  $JAVA -classpath $CLASSPATH $CLASS $PROPERTIES_FILE $LOG_FILE
  procd_set_param respawn
  procd_close_instance
}

stop_service() {
  return 0
}
```

パッケージ名と同名の network-reachability-test-java スクリプトでは下記を行っています.

- 1 行目には「**#!/bin/sh /etc/rc.common**」と記述する
- **USE_PROCD** 変数を定義する
- **start_serivce** シェル関数と **stop_service** シェル関数を定義する
- start_service シェル関数
  - *procd_open_instance* を最初に記述する

  - *procd_set_param command* でアプリケーションの起動コマンドを設定する
  - *procd_set_param respawn* を記述する[*1]
  - *procd_close_instance* を最後に記述する

上記のようにすることで, Java アプリケーションを Procd system manager に登録でき, Java アプリケーションをデーモンとして実行できます.

この例では実装していませんが、stop_service シェル関数は下記のように使用します.

- stop_service シェル関数
  - アプリケーション実行中に作成した一時ファイルなどがあれば, それらを削除する処理を実装する

パッケージ作成用 Makefile は下記です.

リスト 10　パッケージ作成用 Makefile

```
ROOSTER_TOP_DIR ?= $(HOME)/RoosterOS-SDK

ADD_ON_PKG_NAME := network-reachability-test-java
ADD_ON_PKG_VERSION := 1.0
ADD_ON_PKG_DEPENDS := java-runtime
ADD_ON_PKG_RUNTIME_DEPENDS := java-runtime
ADD_ON_PKG_MAINTAINER := your-name@example.com
ADD_ON_PKG_DESCRIPTION := network reachability test application

include $(ROOSTER_TOP_DIR)/mk/add-on-package.mk

NetworkReachabilityTest.class: NetworkReachabilityTest.java
        javac $<

contents: $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) NetworkReachabilityTest.class
        cp NetworkReachabilityTest.class $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)
        cp NetworkReachabilityTest\$$1.class $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)
        mkdir -p $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin
        $(call replace_add_on_keyword,\
                $(ADD_ON_PKG_NAME),\
                $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin/$(ADD_ON_PKG_NAME))
        chmod +x $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin/$(ADD_ON_PKG_NAME)
        cp network-reachability-test-java.properties \
                $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)
        touch $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED)


$(eval $(DefaultTarget))
```

[*1] この記述があるとアプリケーションが何らかの理由で終了したとき, システムが再度 *procd_set_param command* で設定されたアプリケーションの起動コマンドを実行します

include ディレクティブで SDK の add-on-package.mk を取り込みます.

NetworkReachabilityTest.class ターゲットで NetworkReachabilityTest.java をコンパイルします.

NetworkReachabilityTest.java ソースコードが必要です. トップディレクトリに NetworkReachabilityTest.java ソースコードを配置してください.

contents ターゲットでは下記を行っています.

- クラスファイルを$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) ディレクトリにコピーする
- $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin ディレクトリを作成する
- network-reachability-test-java スクリプト中のキーワード[*2] を replace_add_on_keyword マクロ関数で置換して$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin/network-reachability-test-java スクリプトを作成する
- network-reachability-test-java.properties ファイルを$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) ディレクトリにコピーする
- $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED) ファイルを作成する

ディレクトリ構成は下記です.

```
トップ・ディレクトリ/
|
+-- rpk/
|     |
|     +-- CONTROL/
|     |       |
|     |       +-- control ファイル
|     |
|     +-- appctl スクリプト
|
+-- Makefile
|
+-- NetworkReachabilityTest.java
|
+-- network-reachability-test-java スクリプト
|
+-- network-reachability-test-java.properties ファイル
```

make rpk コマンドを実行して network-reachability-test-java_1.0.rpk パッケージを作成し, NSX7000 にインストールしてください.

NSX7000 で**/etc/init.d/rooster_os_application restart** コマンドを実行し, アプリケーションの動作を確認してください.

[*2] @ADD_ON_PKG_NAME@などの「@」で囲まれたもの

- /app/var/network-reachability-test-java.log に定期的にログが記録される

下記以外の方法でアプリケーションのプロセスが終了すると, Procd system manager がアプリケーションを再起動することも確認してください.

- **/etc/init.d/rooster_os_applicatiion stop** コマンド
- **/app/package/network-reachability-test-java/sbin/appctl stop** コマンド
- **/app/package/network-reachability-test-java/sbin/network-reachability-test-java stop** コマンド

Procd system manager によるアプリケーションの再起動を確認する手順は下記のとおりです.

1. ps コマンドでアプリケーションのプロセス ID を確認する

```
root@NSX:~# /usr/bin/ps -e -o pid,args | grep network-reachability-test-java
8122 /app/package/java-runtime/bin/java -classpath /app/package/network-
↪reachability-test-java NetworkReachabilityTest /app/package/network-
↪reachability-test-java/network-reachability-test-java.properties /app/var/
↪network-reachability-test-java.log
9388 grep network-reachability-test-java
```

2. プロセスに TERM シグナルを送信し, ps コマンドでアプリケーションのプロセスが見つからないことを確認する

```
root@NSX:~# kill -TERM 8122
root@NSX:~# /usr/bin/ps -e -o pid,args | grep network-reachability-test-java
9422 grep network-reachability-test-java
```

3. 5 秒ほど待って, ps コマンドでアプリケーションのプロセスが見つかることを確認する

```
root@NSX:~# /usr/bin/ps -e -o pid,args | grep network-reachability-test-java
9721 /app/package/java-runtime/bin/java -classpath /app/package/network-
↪reachability-test-java NetworkReachabilityTest /app/package/network-
↪reachability-test-java/network-reachability-test-java.properties /app/var/
↪network-reachability-test-java.log
9729 grep network-reachability-test-java
```

第 3 章

C/C++ アプリケーションをパッケージ化する

本章では C/C++ アプリケーションをパッケージ化する方法について説明します.

まず, C で HelloWorld アプリケーションを作成し, 手動で NSX 上で実行する方法について説明します. その後, その HelloWorld アプリケーションをパッケージ化します.

つぎに, C++ でネットワーク到達性を確認するアプリケーションを作成して, パッケージ化します. このアプリケーションをデーモンにする方法について説明します.

3.1 C アプリケーションを手動で実行する

NSX7000 で C アプリケーションを手動で実行する手順は下記のとおりです.

1. NSX7000 に「/app/package/**アプリケーション名**」ディレクトリを作成する
2. scp コマンドで C アプリケーションと関連ファイルを「/app/package/**アプリケーション名**」ディレクトリ以下にコピーする
3. NSX7000 に SSH ログインし, C アプリケーションを実行する

以下では NSX7000 で HelloWorld アプリケーションを実行する例を示します.

3.1.1 例: NSX7000 で HelloWorld アプリケーションを実行する

hello-world 実行可能ファイルを準備する

1. HelloWorld アプリケーションのソースコード hello-world.c を作成する

リスト 1　hello-world.c

```
#include <stdio.h>

int main(int argc, char *argv[])
{
     puts("Hello World");
     return 0;
}
```

2. 環境変数を設定する

```
$ STAGING_DIR=${HOME}/RoosterOS-SDK/openwrt/staging_dir/toolchain-arm_cortex-
↪a9+neon_gcc-5.3.0_glibc-2.22_eabi
$ export STAGING_DIR
$ PATH=${STAGING_DIR}/bin:$PATH
$ export PATH
```

3. arm-openwrt-linux-gcc コマンドで hello-world.c をビルドし, hello-world 実行可能ファイルを作成する

```
$ arm-openwrt-linux-gcc -o hello-world hello-world.c
```

NSX7000 にアプリケーション用ディレクトリを準備する

1. **suncorp** アカウントで NSX7000(IP アドレス: 192.168.62.1) に SSH ログインする

```
$ ssh suncorp@192.168.62.1
```

2. **su** コマンドを使って **root** アカウントに切り替える

```
suncorp@NSX:~$ su
Password:


BusyBox v1.24.2 () built-in shell (ash)

root@NSX:/home/suncorp#
```

3. HelloWorld アプリケーション用ディレクトリ/app/package/hello-world を作成する

```
root@NSX:/home/suncorp# mkdir /app/package/hello-world
```

4. **suncorp** アカウントで書き込めるように作成したディレクトリのグループを変更する

```
root@NSX:/home/suncorp# chgrp suncorp /app/package/hello-world
```

5. NSX7000 からログアウトする

```
root@NSX:/home/suncorp# exit
suncorp@NSX:~$ exit
Connection to 192.168.62.1 closed.
```

hello-world 実行可能ファイルを配置する

1. scp コマンドを使って NSX7000 の/app/package/hello-world ディレクトリに hello-world 実行可能ファイルをコピーする

```
$ target_dir=/app/package/hello-world/
$ scp hello-world suncorp@192.168.62.1:$target_dir
hello-world                                   100%   45KB  44.7KB/s   00:00
```

HelloWorld アプリケーションを実行する

1. **suncorp** アカウントで NSX7000(IP アドレス: 192.168.62.1) に SSH ログインする

```
$ ssh suncorp@192.168.62.1
```

2. HelloWorld アプリケーションを実行する

```
suncorp@NSX:~$ app_dir=/app/package/hello-world
suncorp@NSX:~$ ${app_dir}/hello-world
Hello World
```

3.2 C アプリケーションをパッケージ化する

C アプリケーションをパッケージ化する手順は下記のとおりです.

1. 規定のディレクトリと control ファイルを作成する
2. appctl スクリプトを作成する
3. パッケージ作成用 Makefile を作成する
4. パッケージを作成する

3.2.1 規定のディレクトリと control ファイルを作成する

C アプリケーションをパッケージ化するには下記のディレクトリ構成と control ファイル, appctl スクリプト, パッケージ作成用 Makefile が必要です.

```
トップ・ディレクトリ/
|
+-- rpk/
|      |
|      +-- CONTROL/
|      |       |
|      |       +-- control ファイル
|      |
|      +-- appctl スクリプト
|
+-- Makefile
```

トップ・ディレクトリ名は任意です. 以下ではトップ・ディレクトリ名を hello-world として説明します.

1. hello-world トップ・ディレクトリ, rpk サブディレクトリ, CONTROL サブディレクトリを作成します.

```
$ mkdir -p hello-world/rpk/CONTROL
```

2. hello-world/rpk/CONTROL サブディレクトリに下記の control ファイルを作成します.

リスト 2　control ファイル

```
Package: @ADD_ON_PKG_NAME@
Version: @ADD_ON_PKG_VERSION@
Depends: @ADD_ON_PKG_DEPENDS@
Runtime-Depends: @ADD_ON_PKG_RUNTIME_DEPENDS@
Maintainer: @ADD_ON_PKG_MAINTAINER@
Architecture: @ADD_ON_PKG_ARCHITECTURE@
Provides: @ADD_ON_PKG_PROVIDES@
```

(次のページに続く)

(前のページからの続き)

```
Replaces: @ADD_ON_PKG_REPLACES@
Description: @ADD_ON_PKG_DESCRIPTION@
```

3.2.2 appctl スクリプトを作成する

appctl スクリプトを作成します.

appctl スクリプトとは下記の仕様に準拠するシェル・スクリプトです.

- 「start」サブコマンドを理解する
- 「stop」サブコマンドを理解する
- 「restart」サブコマンドを理解する

システムの起動時に **start** サブコマンドが指定されて appctl スクリプトが実行されます. start サブコマンド指定時にアプリケーションを起動する処理を実装します.

システムの終了時に **stop** サブコマンドが指定されて appctl スクリプトが実行されます. stop サブコマンド指定時にアプリケーションを終了する処理を実装します.

アプリケーションを再起動する際に **restart** サブコマンドが指定されて appctl スクリプトが実行されます. restart サブコマンド指定時にアプリケーションを再起動する処理を実装します.

ここでは下記の appctl スクリプトを用意します.

リスト 3　appctl スクリプト

```
#!/bin/sh

PACKAGE_NAME=@ADD_ON_PKG_NAME@
PACKAGE_DIR=/app/package

OUTPUT_FILE=/app/var/hello-world-c.txt

start_app() {
  ${PACKAGE_DIR}/${PACKAGE_NAME}/bin/hello-world >$OUTPUT_FILE
}

stop_app() {
  rm -f $OUTPUT_FILE
}

case "$1" in
  start)
    start_app
    ;;
```

(次のページに続く)

(前のページからの続き)

```
  stop)
    stop_app
    ;;
  restart)
    stop_app
    start_app
    ;;
   *)
    ;;
esac

exit 0
```

標準出力への出力は/app/var/hello-world-c.txt にリダイレクトしています.

3.2.3 パッケージ作成用 Makefile を作成する

パッケージ作成用 Makefile を作成します.

ここでは下記の Makefile を用意します.

リスト 4　パッケージ作成用 Makefile

```
ROOSTER_TOP_DIR ?= $(HOME)/RoosterOS-SDK

ADD_ON_PKG_NAME := hello-world-c
ADD_ON_PKG_VERSION := 1.0
ADD_ON_PKG_MAINTAINER := your-name@example.com
ADD_ON_PKG_DESCRIPTION := hello world application

include $(ROOSTER_TOP_DIR)/mk/add-on-package.mk

hello-world: hello-world.c
        $(CROSS_COMPILE)gcc -o $@ $<

contents: $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) hello-world
        mkdir -p $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin
        cp hello-world $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin
        touch $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED)


$(eval $(DefaultTarget))
```

ADD_ON_PKG_NAME　パッケージ名を記述します. ここでは hello-world-c としています.

ADD_ON_PKG_VERSION　パッケージのバージョンを記述します. ここでは 1.0 としています.

ADD_ON_PKG_MAINTAINER　開発者のメールアドレスを記述します. ここでは your-name@example.com としています.

ADD_ON_PKG_DESCRIPTION　パッケージの説明文を記述します. ここでは「hello world application」としています.

include ディレクティブで SDK の add-on-package.mk を取り込みます. add-on-package.mk を取り込むことで後述の **rpk** ターゲットを利用できるようになります.

hello-world ターゲットで hello-world.c ソースコードをコンパイルし, hello-world 実行可能ファイルをビルドします.

hello-world.c ソースコードが必要です. トップディレクトリに hello-world.c ソースコードを配置してください. hello-world.c を含めたディレクトリ構成は下記です.

```
トップ・ディレクトリ/
|
+-- rpk/
|      |
|      +-- CONTROL/
|      |       |
|      |       +-- control ファイル
|      |
|      +-- appctl スクリプト
|
+-- Makefile
|
+-- hello-world.c
```

contents ターゲットで$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin ディレクトリを作成します.　hello-world 実行可能ファイルを$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin ディレクトリにコピーし, $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED) ファイルを作成します. $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) ディレクトリにコピーしたものがパッケージの内容物になります.

3.2.4 パッケージを作成する

make rpk コマンドを実行してパッケージを作成します.

```
$ make rpk
```

成功すると **hello-world-c_1.0.rpk** ファイルが作成されます.

下記の手順でインストールしてください.

1. scp コマンドを使って NSX7000 の/tmp ディレクトリに hello-world-c_1.0.rpk をコピーする

```
$ scp hello-world-c_1.0.rpk suncorp@192.168.62.1:/tmp
hello-world-c_1.0.rpk                         100% 2807     2.7KB/s   00:00
```

2. suncorp アカウントで NSX7000(IP アドレス: 192.168.62.1) に SSH ログインする

```
$ ssh suncorp@192.168.62.1
```

3. su マンドを使って root アカウントに切り替える

```
suncorp@NSX:~$ su
Password:


BusyBox v1.24.2 () built-in shell (ash)

root@NSX:/home/suncorp#
```

4. **rpkg install** コマンドを実行して hello-world-c_1.0.rpk をインストールする

```
root@NSX:/home/suncorp# rpkg install /tmp/hello-world-c_1.0.rpk
UBI device number 4, total 768 LEBs (97517568 bytes, 93.0 MiB),
available 0 LEBs (0 bytes), LEB size 126976 bytes (124.0 KiB)
Installing hello-world-c (1.0) on root
Configuring hello-world-c.
Installing hello-world-c (1.0) on root
Configuring hello-world-c.
```

5. **/etc/init.d/rooster_os_application restart** コマンドを実行してアプリケーションを再起動する

```
root@NSX:/home/suncorp# /etc/init.d/rooster_os_application restart
app.img: OK
app.img: OK
```

実行後に下記を確認してください.

- /app/package/hello-world-c ディレクトリが作成されている
- /app/package/hello-world-c ディレクトリに bin/hello-world 実行可能ファイルがある
- /app/package/hello-world-c ディレクトリに sbin/appctl スクリプトがある
- /app/var/hello-world-c.txt ファイルが作成されている
- /app/var/hello-world-c.txt ファイルに「Hello World」と書き込まれている

アプリケーションを停止したときに何が起きるかを確認するため**/etc/init.d/rooster_os_application stop** コマンドを実行してください.

```
root@NSX:/home/suncorp# /etc/init.d/rooster_os_application stop
```

実行後に下記を確認してください.

- /app/package/hello-world-c ディレクトリが削除されている
- /app/var/hello-world-c.txt ファイルが削除されている

再度アプリケーションを開始するには**/etc/init.d/rooster_os_application start** コマンドを実行してください.

```
root@NSX:/home/suncorp# /etc/init.d/rooster_os_application start
app.img: OK
app.img: OK
```

3.3 C/C++ アプリケーションをデーモン化する

C/C++ アプリケーションをデーモン化する手順は下記のとおりです.

- パッケージ名と同名のスクリプトを作成する
- appctl からパッケージ名と同名のスクリプトを実行する
- appctl から実行されるスクリプトで, C/C++ アプリケーションを Procd system manager に登録する処理を実装する

本節では例としてネットワーク到達性を定期的に確認する C++ アプリケーションを作成します. この C++ アプリケーションを Procd system manager に登録するスクリプトを実装し, C++ アプリケーションをデーモン化します.

注釈: Procd system manager がデーモン化処理を行なうので C/C++ アプリケーション内でデーモン化処理を実装する必要はありません.

また, Procd system manager を利用する場合は C/C++ アプリケーション内でデーモン化処理を行わないでください.

もし C/C++ アプリケーション内でデーモン化処理を行なうと, Procd system manager は C/C++ アプリケーション・プロセスを見失い, プロセスが終了したものと判断してしまいます.

外部ライブラリとして Poco[*1] を使用します.

3.3.1 Poco のビルド環境を用意する

1. トップ・ディレクトリに poco サブディレクトリを作成し, poco サブディレクトリに移動する

```
$ mkdir poco
$ cd poco
```

2. poco-1.9.0.tar.gz をダウンロードする

```
$ curl -O https://pocoproject.org/releases/poco-1.9.0/poco-1.9.0.tar.gz
```

3. poco-1.9.0.tar.gz を展開する

```
$ tar xvf poco-1.9.0.tar.gz
```

4. 展開によって作成された poco-1.9.0 ディレクトリを src ディレクトリにリネームする

[*1] https://pocoproject.org/

```
$ mv poco-1.9.0 src
```

5. Makefile を作成する

リスト 5　Poco ビルド用 Makefile

```
ROOSTER_TOP_DIR ?= $(HOME)/RoosterOS-SDK

include $(ROOSTER_TOP_DIR)/mk/develop.mk

POCO_INSTALL_DIR ?= $(ROOSTER_PACKAGE_TMP_DIR)/install

$(ROOSTER_PACKAGE_CONFIGURED):
        $(call cmake_configure_package)

$(ROOSTER_PACKAGE_BUILT):
        $(call build_package)

$(ROOSTER_PACKAGE_INSTALLED):
        $(call install_package,DESTDIR=$(POCO_INSTALL_DIR))

$(eval $(DefaultTarget))
```

6. トップ・ディレクトリに戻る

```
$ cd ..
```

3.3.2 例: ネットワーク到達性を定期的に確認する C++ アプリケーションをデーモン化する

ネットワーク到達性を定期的に確認する C++ アプリケーションを作成します．パッケージ名は network-reachability-test-cpp とし, 本パッケージの仕様を下記とします.

- ネットワーク到達性を確認する送信先ホストは ini ファイルで指定する
  - キーは *target_ipv4_address* とする
  - 値は IPv4 アドレスで指定する
- ネットワーク到達性を確認する ICMP echo リクエストの送信間隔は ini ファイルで指定する
  - キーは *interval* とする
  - 値は非負の数値（単位は秒）で指定する
- ネットワーク到達性を確認する ICMP echo リクエストのタイムアウト値は ini ファイルで指定する
  - キーは *timeout* とする

  - 値は非負の数値（単位は秒）で指定する

- 第 1 引数に ini ファイルのパスを指定する
- 第 2 引数にログファイルのパスを指定する

上記にもとづいて作成した C++ のソースコードが network-reachability-test.cc です.

リスト 6　network-reachability-test.cc

```
#include "Poco/Format.h"
#include "Poco/Message.h"
#include "Poco/FileChannel.h"
#include "Poco/PatternFormatter.h"
#include "Poco/FormattingChannel.h"
#include "Poco/Thread.h"
#include "Poco/Net/ICMPClient.h"
#include "Poco/Net/SocketAddress.h"
#include "Poco/Util/IniFileConfiguration.h"
#include "Poco/Exception.h"
#include "Poco/Net/NetException.h"
#include <iostream>
#include <string>
#include <utility>
#include <cstdlib>

namespace {
    // プロパティ名の定義
    const char INI_FILE_KEY_TARGET_IPV4_ADDRESS[] = "target_ipv4_address";
    const char INI_FILE_KEY_INTERVAL[] = "interval";
    const char INI_FILE_KEY_TIMEOUT[] = "timeout";

    // プロパティのデフォルト値
    const char DEFAULT_TARGET_IPV4_ADDRESS[] = "127.0.0.1";
    const int DEFAULT_INTERVAL = 10;
    const int DEFAULT_TIMEOUT = 10;

    // Ping 設定
    const int PING_DATA_SIZE = 48;
    const int PING_TTL = 128;
    const int PING_REPEAT = 1;

    // ログファイル設定
    const char LOG_SOURCE[] ="network-reachability-test";
    const char LOG_FORMAT[] = "%s: %Y-%m-%d %H:%M:%S: %t";
    Poco::FormattingChannel* logger;

    // 指定された ini_file からキー name に対応する SocketAddress を取得する.
    // 取得できなければ default_value から SocketAddress を作成する
    Poco::Net::SocketAddress
```

(次のページに続く)

(前のページからの続き)

```
getIPv4Address(Poco::Util::IniFileConfiguration& ini_file,
               const std::string& name,
               const std::string& default_value)
{
    const Poco::Net::SocketAddress::Family family =
        Poco::Net::SocketAddress::Family::IPv4;
    std::string value(ini_file.getString(name, default_value));

    try {
        return Poco::Net::SocketAddress(family, value, 0);
    } catch (Poco::Net::HostNotFoundException& e) {
        std::string text(Poco::format("ini file: '%s' is invalid: %s",
                                      name, e.displayText()));
        Poco::Message log_message(LOG_SOURCE,
                                  text,
                                  Poco::Message::Priority::PRIO_WARNING);
        logger->log(log_message);
        log_message.setText(Poco::format("use default value: '%s'",
                                         default_value));
        logger->log(log_message);
        return Poco::Net::SocketAddress(family, default_value, 0);
    } catch (Poco::Net::AddressFamilyMismatchException& e) {
        std::string text(Poco::format("ini file: '%s' is invalid: %s",
                                      name, e.displayText()));
        Poco::Message log_message(LOG_SOURCE,
                                  text,
                                  Poco::Message::Priority::PRIO_WARNING);
        logger->log(log_message);
        log_message.setText(Poco::format("use default value: '%s'",
                                         default_value));
        logger->log(log_message);
        return Poco::Net::SocketAddress(family, default_value, 0);
    }
}

// ini ファイルから"target_ipv4_address"キーの値を SocketAddress として取得する
Poco::Net::SocketAddress
getTargetIPv4Address(Poco::Util::IniFileConfiguration& ini_file)
{
    return getIPv4Address(ini_file,
                          INI_FILE_KEY_TARGET_IPV4_ADDRESS,
                          DEFAULT_TARGET_IPV4_ADDRESS);
}

// 指定された ini_file からキー name に対応する int を取得する.
// 取得できない, もしくはその値が 0 以下なら default_value から int を作成する
int getPositiveIntValue(Poco::Util::IniFileConfiguration& ini_file,
                        const std::string& name,
```

(次のページに続く)

(前のページからの続き)

```
                            int default_value)
{
    int value = -1;
    try {
        value = ini_file.getInt(name, default_value);
    } catch (Poco::SyntaxException& e) {
        std::string text(Poco::format(
                            "ini file: '%s' is not an int value: %s",
                            name, e.displayText()));
        Poco::Message log_message(LOG_SOURCE,
                                  text,
                                  Poco::Message::Priority::PRIO_WARNING);
        logger->log(log_message);
        log_message.setText(Poco::format("use default value: '%d'",
                                         default_value));
        logger->log(log_message);
        return default_value;
    }

    if (value <= 0) {
        std::string text(Poco::format(
                            "ini file: '%s' is not a positive int value: '%d'",
                            value));
        Poco::Message log_message(LOG_SOURCE,
                                  text,
                                  Poco::Message::Priority::PRIO_WARNING);
        logger->log(log_message);
        log_message.setText(Poco::format("use default value: '%d'",
                                         default_value));
        logger->log(log_message);
        return default_value;
    }

    return value;
}

// ini ファイルから"interval"キーの値を int として取得する
int getInterval(Poco::Util::IniFileConfiguration& ini_file)
{
    return getPositiveIntValue(ini_file,
                               INI_FILE_KEY_INTERVAL,
                               DEFAULT_INTERVAL) * 1000;
}

// ini ファイルから"timeout"キーの値を int として取得する
int getTimeout(Poco::Util::IniFileConfiguration& ini_file)
{
    return getPositiveIntValue(ini_file,
```

(次のページに続く)

(前のページからの続き)

```
                                    INI_FILE_KEY_TIMEOUT,
                                    DEFAULT_TIMEOUT) * 1000;
    }
}

int main(int argc, char* argv[])
{
    if (argc < 3) {
        std::cerr << "need ini file path and log file path" << std::endl;
        return EXIT_FAILURE;
    }

    std::string ini_file_path(argv[1]); // ini ファイルへのパス
    std::string log_file_path(argv[2]); // ログファイルへのパス

    // ログファイルを開く
    Poco::AutoPtr<Poco::FileChannel> file_channel;
    Poco::AutoPtr<Poco::PatternFormatter> pattern_formatter;
    Poco::AutoPtr<Poco::FormattingChannel> fmt_channel;

    file_channel.assign(new Poco::FileChannel(log_file_path));
    pattern_formatter.assign(new Poco::PatternFormatter(LOG_FORMAT));
    pattern_formatter->setProperty("times", "local");
    fmt_channel.assign(new Poco::FormattingChannel(pattern_formatter.get(),
                                                   file_channel.get()));
    try {
        fmt_channel->open();
    } catch (Poco::Exception& e) {
        std::cerr <<
            Poco::format("failed to open '%s': %s",
                         log_file_path, e.displayText()) <<
            std::endl;
        return EXIT_FAILURE;
    }

    logger = fmt_channel.get();

    // ini ファイルを読み込む
    Poco::AutoPtr<Poco::Util::IniFileConfiguration> ini_file(
        new Poco::Util::IniFileConfiguration());
    try {
        ini_file->load(ini_file_path);
    } catch (Poco::Exception& e) {
        std::cerr <<
            Poco::format("failed to open '%s': %s",
                         ini_file_path, e.displayText()) <<
            std::endl;
        return EXIT_FAILURE;
```

(次のページに続く)

(前のページからの続き)

```
    }

    // ini ファイルからターゲットの IPv4 アドレス, 送信間隔,
    // タイムアウトの設定値を取得する
    Poco::Net::SocketAddress target_ipv4_addr(getTargetIPv4Address(*ini_file));
    std::string target_ipv4_addr_str(target_ipv4_addr.host().toString());
    int interval = getInterval(*ini_file);
    int timeout = getTimeout(*ini_file);

    Poco::Message log_message;
    log_message.setSource(LOG_SOURCE);
    log_message.setPriority(Poco::Message::Priority::PRIO_INFORMATION);

    // 送信間隔ごとにネットワーク到達性を確認する
    while (true) {
        std::string text;

        try {
            Poco::Net::ICMPClient::ping(target_ipv4_addr,
                                        Poco::Net::SocketAddress::Family::IPv4,
                                        PING_REPEAT,
                                        PING_DATA_SIZE,
                                        PING_TTL,
                                        timeout);
            text = Poco::format("'%s' is reachable", target_ipv4_addr_str);
        } catch (Poco::Exception& e) {
            text = Poco::format("'%s' is unreachable: %s",
                                target_ipv4_addr_str, e.displayText());
        }

        log_message.setText(text);
        logger->log(log_message);

        Poco::Thread::sleep(interval);
    }

    return 0;
}
```

ini ファイル network-reachability-test-cpp.ini は下記です.

リスト 7　network-reachability-test-cpp.ini

```
target_ipv4_address = 192.168.62.100
interval = 5
timeout = 30
```

control ファイルは下記です. hello-world-c パッケージと同一です.

リスト 8　control ファイル

```
Package: @ADD_ON_PKG_NAME@
Version: @ADD_ON_PKG_VERSION@
Depends: @ADD_ON_PKG_DEPENDS@
Runtime-Depends: @ADD_ON_PKG_RUNTIME_DEPENDS@
Maintainer: @ADD_ON_PKG_MAINTAINER@
Architecture: @ADD_ON_PKG_ARCHITECTURE@
Provides: @ADD_ON_PKG_PROVIDES@
Replaces: @ADD_ON_PKG_REPLACES@
Description: @ADD_ON_PKG_DESCRIPTION@
```

appctl スクリプトは下記です.

リスト 9　appctl スクリプト

```
#!/bin/sh

PACKAGE_NAME=@ADD_ON_PKG_NAME@
PACKAGE_DIR=/app/package

exec $PACKAGE_DIR/$PACKAGE_NAME/sbin/$PACKAGE_NAME $@
```

appctl スクリプトから実行される network-reachability-test-cpp スクリプトは下記です.

リスト 10　network-reachability-test-cpp スクリプト

```
#!/bin/sh /etc/rc.common

USE_PROCD=1

PACKAGE_NAME=@ADD_ON_PKG_NAME@
PACKAGE_DIR=/app/package

LD_LIBRARY_PATH=${PACKAGE_DIR}/${PACKAGE_NAME}/lib
COMMAND=${PACKAGE_DIR}/${PACKAGE_NAME}/bin/network-reachability-test
INI_FILE=${PACKAGE_DIR}/${PACKAGE_NAME}/etc/${PACKAGE_NAME}.ini
LOG_FILE=/app/var/${PACKAGE_NAME}.log

start_service() {
  procd_open_instance
  procd_set_param env LD_LIBRARY_PATH=$LD_LIBRARY_PATH
  procd_set_param command $COMMAND $INI_FILE $LOG_FILE
  procd_set_param respawn
  procd_close_instance
}

stop_service() {
```

(次のページに続く)

(前のページからの続き)

```
  return 0
}
```

パッケージ名と同名の network-reachability-test-cpp スクリプトでは下記を行っています.

- 1 行目には「**#!/bin/sh /etc/rc.common**」と記述する
- **USE_PROCD** 変数を定義する
- **start_serivce** シェル関数と **stop_service** シェル関数を定義する
- start_service シェル関数
  - *procd_open_instance* を最初に記述する
  - *procd_set_param env* で動的にリンクするライブラリのパスを設定する
  - *procd_set_param command* でアプリケーションの起動コマンドを設定する
  - *procd_set_param respawn* を記述する[*2]
  - *procd_close_instance* を最後に記述する

上記のようにすることで, C++ アプリケーションを Procd system manager に登録でき, C++ アプリケーションをデーモンとして実行できます.

この例では実装していませんが、stop_service シェル関数は下記のように使用します.

- stop_service シェル関数
  - アプリケーション実行中に作成した一時ファイルなどがあれば, それらを削除する処理を実装する

パッケージ作成用 Makefile は下記です.

リスト 11　パッケージ作成用 Makefile

```
ROOSTER_TOP_DIR ?= $(HOME)/RoosterOS-SDK

ADD_ON_PKG_NAME := network-reachability-test-cpp
ADD_ON_PKG_VERSION := 1.0
ADD_ON_PKG_MAINTAINER := your-name@example.com
ADD_ON_PKG_DESCRIPTION := network reachability test application

include $(ROOSTER_TOP_DIR)/mk/add-on-package.mk

CMAKE_INSTALL_PREFIX := $(ADD_ON_PACKAGE_MOUNT_POINT_DIR)
export CMAKE_INSTALL_PREFIX
```

(次のページに続く)

[*2] この記述があるとアプリケーションが何らかの理由で終了したとき, システムが再度　*procd_set_param command* で設定されたアプリケーションの起動コマンドを実行します

(前のページからの続き)

```
POCO_INSTALL_DIR := $(CURDIR)/develop
export POCO_INSTALL_DIR

INCLUDE_DIR := $(POCO_INSTALL_DIR)/$(CMAKE_INSTALL_PREFIX)/include
LIB_DIR := $(POCO_INSTALL_DIR)/$(CMAKE_INSTALL_PREFIX)/lib
LIBS := -lPocoUtil -lPocoXML -lPocoJSON -lPocoNet -lPocoFoundation

$(INCLUDE_DIR)/Poco/Version.h:
        make -C poco install

poco: $(INCLUDE_DIR)/Poco/Version.h

network-reachability-test: network-reachability-test.cc poco
        $(CROSS_COMPILE)g++ -std=c++14 -I$(INCLUDE_DIR) -L$(LIB_DIR) -o $@ $< $(LIBS)

contents: $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) network-reachability-test
        mkdir -p $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/lib
        cp -a $(LIB_DIR)/*.so* $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/lib
        mkdir -p $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin
        cp network-reachability-test $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin
        mkdir -p $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin
        $(call replace_add_on_keyword,\
                $(ADD_ON_PKG_NAME),\
                $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin/$(ADD_ON_PKG_NAME))
        chmod +x $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin/$(ADD_ON_PKG_NAME)
        mkdir -p $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/etc
        cp network-reachability-test-cpp.ini $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/etc
        touch $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED)


$(eval $(DefaultTarget))
```

include ディレクティブで SDK の add-on-package.mk を取り込みます.

$(INCLUDE_DIR)/Poco/Version.h ターゲットで Poco をビルドします.

network-reachability-test ターゲットで network-reachability-test.cc ソースコードをコンパイルし, network-reachability-test 実行可能ファイルをビルドします.

network-reachability-test.cc ソースコードが必要です. トップディレクトリに network-reachability-test.cc ソースコードを配置してください.

contents ターゲットでは下記を行っています.

- $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/lib ディレクトリを作成する
- Poco ライブラリを$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/lib ディレクトリにコピーする

- $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin ディレクトリを作成する
- network-reachability-test 実行可能ファイルを$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/bin ディレクトリにコピーします.
- $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin ディレクトリを作成する
- network-reachability-test-cpp スクリプト中のキーワード[*3] を replace_add_on_keyword マクロ関数で置換して$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/sbin/network-reachability-test-cpp スクリプトを作成する
- $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/etc ディレクトリを作成する
- network-reachability-test-cpp.ini ファイルを$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)/etc ディレクトリにコピーする
- $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED) ファイルを作成する

ディレクトリ構成は下記です.

```
トップ・ディレクトリ/
|
+-- poco/
|     |
|     +-- src/ : poco-1.9.0 のソースコード
|     |
|     +-- Makefile: Poco ビルド用 Makefile
|
+-- rpk/
|     |
|     +-- CONTROL/
|     |       |
|     |       +-- control ファイル
|     |
|     +-- appctl スクリプト
|
+-- Makefile
|
+-- network-reachability-test.cc
|
+-- network-reachability-test-cpp スクリプト
|
+-- network-reachability-test-cpp.ini ファイル
```

make rpk コマンドを実行して network-reachability-test-cpp_1.0.rpk パッケージを作成し, NSX7000 にインストールしてください.

[*3] @ADD_ON_PKG_NAME@などの「@」で囲まれたもの

NSX7000 で**/etc/init.d/rooster_os_application restart** コマンドを実行し, アプリケーションの動作を確認してください.

- /app/var/network-reachability-test-cpp.log に定期的にログが記録される

下記以外の方法でアプリケーションのプロセスが終了すると, Procd system manager がアプリケーションを再起動することも確認してください.

- **/etc/init.d/rooster_os_applicatiion stop** コマンド
- **/app/package/network-reachability-test-cpp/sbin/appctl stop** コマンド
- **/app/package/network-reachability-test-cpp/sbin/network-reachability-test-cpp stop** コマンド

Procd system manager によるアプリケーションの再起動を確認する手順は下記のとおりです.

1. ps コマンドでアプリケーションのプロセス ID を確認する

```
root@NSX:~# /usr/bin/ps -e -o pid,args | grep network-reachability-test-cpp
11489 /app/package/network-reachability-test-cpp/bin/network-reachability-test /
↪app/package/network-reachability-test-cpp/etc/network-reachability-test-cpp.ini␣
↪/app/var/network-reachability-test-cpp.log
11531 grep network-reachability-test-cpp
```

2. プロセスに TERM シグナルを送信し, ps コマンドでアプリケーションのプロセスが見つからないことを確認する

```
root@NSX:~# kill -TERM 11489
root@NSX:~# /usr/bin/ps -e -o pid,args | grep network-reachability-test-cpp
11593 grep network-reachability-test-cpp
```

3. 5 秒ほど待って, ps コマンドでアプリケーションのプロセスが見つかることを確認する

```
root@NSX:~# /usr/bin/ps -e -o pid,args | grep network-reachability-test-cpp
11600 /app/package/network-reachability-test-cpp/bin/network-reachability-test /
↪app/package/network-reachability-test-cpp/etc/network-reachability-test-cpp.ini␣
↪/app/var/network-reachability-test-cpp.log
11629 grep network-reachability-test-cpp
```

第 4 章

ケーパビリティを指定してアプリケーションを実行する

NSX7000 ではアプリケーションに制限を課しています.

root 権限を持つアプリケーション[*2] でも下記のケーパビリティを必要とするシステムコールは許可されません.

- CAP_MAC_ADMIN
- CAP_MAC_OVERRIDE
- CAP_SETFCAP
- CAP_SYS_PTRACE[*1]
- CAP_SYS_RAWIO
- CAP_AUDIT_CONTROL
- CAP_AUDIT_READ
- CAP_AUDIT_WRITE
- CAP_BLOCK_SUSPEND
- CAP_FSETID
- CAP_LINUX_IMMUTABLE
- CAP_MKNOD
- CAP_NET_ADMIN
- CAP_SYS_ADMIN

[*2] 実効 UID が 0 のプロセスのことです
[*1] 開発者モードが ON なら許可します.

- CAP_SYS_MODULE
- CAP_SYS_NICE
- CAP_SYS_RESOURCE
- CAP_SYSLOG
- CAP_WAKE_ALARM

上記のケーパビリティをもつシステムコールを実行するとそのシステムコールはエラーを表す値を返し, errno に EPERM をセットします.

開発者モードが ON の場合は **rooster-os-cap-add-on** コマンドを使って任意のケーパビリティを指定してコマンドを実行できます.

ただし, 下記のケーパビリティは指定しても無視するため設定できません.

- CAP_MAC_ADMIN
- CAP_MAC_OVERRIDE
- CAP_SETFCAP
- CAP_SYS_RAWIO

rooster-os-cap-add-on コマンドの使用方法は下記のとおりです.

```
root@NSX:~# /safe/bin/rooster-os-cap-add-on -h
usage: /safe/bin/rooster-os-cap-add-on [OPTION] -- command [args...]

optional arguments:
  -h, --help          : show this help message and exit
  -s, --syslog-only   : not write error message to stderr
  -c, --caps=cap-set  : set the process capabilities
                        to those specified by cap-set
```

–caps オプションでケーパビリティを指定します. 下記のような文字列が指定できます.

- –caps=\"all=eip\"
- –caps=\"cap_chown=eip\"
- –caps=\"cap_chown,cap_dac_override,cap_dac_read_search=eip\"

cap-set として指定できる文字列は cap_from_text(3) で指定できる文字列です. 詳細は https://linux.die.net/man/3/cap_from_text を参照してください.

以下では/app/var ディレクトリに usb-memory を作成し, /app/var/usb-memory ディレクトリに USB メモリをマウントするためのパッケージを作成します.

4.1 例: 起動時に USB メモリをマウント, 終了時に USB メモリをアンマウントする

起動時に USB メモリをマウント, 終了時に USB メモリをアンマウントするアプリケーションを作成します.

control ファイルは下記です.

リスト 1　control ファイル

```
Package: @ADD_ON_PKG_NAME@
Version: @ADD_ON_PKG_VERSION@
Depends: @ADD_ON_PKG_DEPENDS@
Runtime-Depends: @ADD_ON_PKG_RUNTIME_DEPENDS@
Maintainer: @ADD_ON_PKG_MAINTAINER@
Architecture: @ADD_ON_PKG_ARCHITECTURE@
Provides: @ADD_ON_PKG_PROVIDES@
Replaces: @ADD_ON_PKG_REPLACES@
Description: @ADD_ON_PKG_DESCRIPTION@
```

appctl スクリプトは下記です.

リスト 2　appctl スクリプト

```
#!/bin/sh

PACKAGE_NAME=@ADD_ON_PKG_NAME@
PACKAGE_DIR=/app/package

MOUNT_POINT=/app/var/usb-memory

start_app() {
  if [ ! -e $MOUNT_POINT ]; then
    mkdir -p $MOUNT_POINT
  fi

  /safe/bin/rooster-os-cap-add-on --caps="cap_sys_admin=eip" -- \
                                  mount /dev/sda1 $MOUNT_POINT
  date > ${MOUNT_POINT}/mounted.txt
}

stop_app() {
  mountpoint -q $MOUNT_POINT
  if [ $? -eq 0 ]; then
    /safe/bin/rooster-os-cap-add-on --caps="cap_sys_admin=eip" -- \
                                    umount $MOUNT_POINT
  fi
}
```

(次のページに続く)

(前のページからの続き)

```
case "$1" in
  start)
    start_app
    ;;
  stop)
    stop_app
    ;;
  restart)
    stop_app
    start_app
    ;;
   *)
    ;;
esac

exit 0
```

起動時 = start サブコマンド指定時に下記を行います.

- /app/var/usb-memory ディレクトリがなければ作成する
- rooster-os-cap-add-on コマンドを使って mount コマンドを実行する. USB メモリ/dev/sda1 を/app/var/usb-memory ディレクトリにマウントする
- マウントに成功した時刻を/app/var/usb-memory/mounted.txt に書き込む

終了時 = stop サブコマンド指定時に下記を行います.

- /app/var/usb-memory ディレクトリに USB メモリがマウントされているかをチェックする
- /app/var/usb-memory ディレクトリに USB メモリがマウントされていれば, rooster-os-cap-add-on コマンドを使って umount コマンドを実行し, USB メモリをアンマウントする

パッケージ作成用 Makefile は下記です.

リスト 3　パッケージ作成用 Makefile

```
ROOSTER_TOP_DIR ?= $(HOME)/RoosterOS-SDK

ADD_ON_PKG_NAME := mount-usb-memory
ADD_ON_PKG_VERSION := 1.0
ADD_ON_PKG_MAINTAINER := your-name@example.com
ADD_ON_PKG_DESCRIPTION := mounting usb-memory application

include $(ROOSTER_TOP_DIR)/mk/add-on-package.mk

contents: $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR)
        touch $(ROOSTER_PACKAGE_ADD_ON_CONTENTS_PREPARED)
```

(次のページに続く)

(前のページからの続き)

```
clean:



$(eval $(DefaultTarget))
```

include ディレクティブで SDK の add-on-package.mk を取り込みます.

appctl スクリプトのみで処理が完結するので, contents ターゲットで用意する内容物はありません.

ディレクトリ構成は下記です.

```
トップ・ディレクトリ/
|
+-- rpk/
|      |
|      +-- CONTROL/
|      |       |
|      |       +-- control ファイル
|      |
|      +-- appctl スクリプト
|
+-- Makefile
```

make rpk コマンドを実行して mount-usb-memory_1.0.rpk パッケージを作成し, NSX7000 にインストールしてください.

USB メモリを挿した NSX7000 で**/etc/init.d/rooster_os_application restart** コマンドを実行し, 下記を確認してください.

- /app/var/usb-memory ディレクトリが存在する
- /app/var/usb-memory/mounted.txt ファイルが存在する
- **mountpoint /app/var/usb-memory** を実行すると下記の出力を得られる

```
root@NSX:~# mountpoint /app/var/usb-memory
/app/var/usb-memory is a mountpoint
```

/etc/init.d/rooster_os_application stop コマンドを実行し, 下記を確認してください.

- /app/var/usb-memory ディレクトリが存在する
- /app/var/usb-memory ディレクトリが空である
- **mountpoint /app/var/usb-memory** を実行すると下記の出力を得られる

```
root@NSX:~# mountpoint /app/var/usb-memory
/app/var/usb-memory is not a mountpoint
```

第 5 章

おわりに

Add-on アプリケーションのパッケージ作成は以上で終了です.

必要に応じて付録を参照してください.

付録 A

RPK ファイル

RPK とは RoosterOS PacKage の略です.

RoosterOS では, インストールやアンインストールするときの一つのまとまり（コマンド, ライブラリ, 設定ファイル など）を**パッケージ (package)** と呼びます.

RoosterOS にパッケージをインストールするときに使用するファイル形式を **RPK ファイル** と呼びます.

RPK ファイルのファイル形式は OpenWrt が採用している ipk ファイルと同じものです. また, ipk ファイルは Debian パッケージファイル (deb ファイル) をベースにしています. したがって下記の関係が成り立ちます:

```
RPK ファイル    ipk ファイル    deb ファイル
```

RPK ファイルは tar ファイルを gzip 圧縮したファイル, いわゆる tar.gz ファイルです. RPK ファイルは下記のファイルを含んでいます.

- debian-binary
- data.tar.gz
- control.tar.gz

RPK ファイル内の debian-binary の内容は下記のとおりです. ipk ファイル や deb ファイルとの互換性の維持を目的に RPK ファイルに含んでいます.:

```
2.0
```

data.tar.gz ファイルは実際に RoosterOS にインストールされるファイルを含みます.

data.tar.gz は下記の 3 つのファイルを含みます.

- app-image/package/**アプリケーション名** /app.img
- app-image/package/**アプリケーション名** /app.img.cksum
- app-image/package/**アプリケーション名** /runtime-dependencies

app.img ファイルは SquashFS ファイルシステムのイメージ・ファイルです. パッケージ作成用 Makefile の contents ターゲットで$(ROOSTER_PACKAGE_ADD_ON_CONTENTS_DIR) ディレクトリにコピーしたファイルを含んでいます.

app.img.cksum ファイルは app.img のチェックサムファイルです. パッケージ作成用 Makefile の rpk ターゲットで sha256sum コマンドを使って作成します.

runtime-dependencies ファイルは実行時の依存関係を記述したファイルです. パッケージ作成用 Makefile の rpk ターゲットで ADD_ON_PKG_RUNTIME_DEPENDS 変数を元に作成します. runtime-dependencies ファイルがどのように使われるかは「*Add-on* **アプリケーションの起動と終了**」を参照してください.

control.tar.gz ファイルはパッケージ情報を記述した control ファイルを含みます. パッケージ作成用 Makefile の rpk ターゲットで ADD_ON_PKG_NAME 変数などの ADD_ON プレフィクスをもつ変数を元に作成します.

付録 B

Add-on アプリケーションの起動と終了

B.1 Add-on アプリケーションの起動

RoosterOS は起動すると/etc/rc.d ディレクトリ以下の **S** で始まるスクリプトをアルファベット順に実行します.

この処理の最後で `/etc/rc.d/S99~rooster_os_application boot` を実行します.

/etc/rc.d/S99~rooster_os_application は/etc/init.d/rooster_os_application へのシンボリックリンクです.

コマンドライン引数として `boot` を指定された`/etc/init.d/rooster_os_application` は下記の処理を行います.

1. Add-on アプリケーションと一緒にインストールされた Add-on アプリケーションごとの実行時依存情報ファイル (runtime-dependencies) をまとめて, 単一の実行時依存情報ファイルを作成します
2. 単一の実行時依存情報ファイルに記述されたパッケージの順に下記の処理を行います
    1. /app-image/package/<パッケージ名>/app.img ファイルがあるかチェックします. app.img ファイルなければ以下の処理をスキップします
    2. /app-image/package/<パッケージ名>/app.img.cksum ファイルの SHA256 チェックサム値と/app-image/package/<パッケージ名>/app.img ファイルの SHA256 チェックサムが一致するか検証します. SHA256 チェックサム値が一致しなければ以下の処理をスキップします
    3. /app/package/<パッケージ名>ディレクトリを作成します. 作成できなければ以下の処理をスキップします
    4. loop デバイスを使って/app-image/package/<パッケージ名>/app.img ファイルを/app/package/<パッケージ名>ディレクトリにマウントします. マウントできなければ以下の処理をスキップします
    5. /app/package/<パッケージ名>/sbin/appctl ファイルが存在し, かつ, 実行可能なら, `/app/package/<`**パッケージ名**`>/sbin/appctl start` を実行します

B.2 Add-on アプリケーションの終了

RoosterOS は終了するときに/etc/rc.d ディレクトリ以下の **K** で始まるスクリプトをアルファベット順に実行します.

この処理の最初に `/etc/rc.d/S99~rooster_os_application shutdown` を実行します.

/etc/rc.d/S99~rooster_os_application は/etc/init.d/rooster_os_application へのシンボリックリンクです.

コマンドライン引数として `shutdown` を指定された`/etc/init.d/rooster_os_application` は下記の処理を行います.

1. Add-on アプリケーションと一緒にインストールされた Add-on アプリケーションごとの実行時依存情報ファイル (runtime-dependencies) をまとめて, 単一の実行時依存情報ファイルを作成します
2. 単一の実行時依存情報ファイルに記述されたパッケージの逆順に下記の処理を行います
    1. /app/package/<パッケージ名>ディレクトリがあるか確認するなければ以下の処理をスキップします
    2. /app/package/<パッケージ名>/sbin/appctl ファイルが存在し, かつ, 実行可能なら, `/app/package/<`**パッケージ名**`>/sbin/appctl stop` を実行します
    3. /app/package/<パッケージ名>ディレクトリに app.img ファイルがマウントされているか確認します. マウントされていればアンマウントします. アンマウントできなければ以下の処理をスキップします
    4. /app/package/<パッケージ名>ディレクトリを削除します

付録 C

パッケージのインストールとアンインストール

パッケージのインストールやアンインストールには **rpkg** コマンドを使用します.

root ユーザで実行する必要があります.

C.1 インストール

パッケージをインストールするには **install** サブコマンドを指定します.

```
rpkg install RPK ファイルのパス名 ...
```

RPK **ファイルのパス名** は複数指定できます.

パッケージのアップグレードも install サブコマンドを使って行います.

インストール済みパッケージと同じバージョンや古いバージョンのパッケージをインストールしようとするとエラーになります.

同じバージョンのパッケージを強制的に再インストールしたい場合は**–force-reinstall** オプションを指定してください.

```
rpkg --force-reinstall install RPK ファイルのパス名 ...
```

インストール時には引数で与えられた RPK ファイルの依存関係を解決し, 依存関係を満たす順序でパッケージをインストールします.

```
例: app-A が app-B に依存し, app-B が app-C に依存している場合

root@NSX:~# rpkg install app-A app-B app-C
-> app-C をインストールする
```

(次のページに続く)

(前のページからの続き)

```
-> app-B をインストールする
-> app-A をインストールする
```

C.2 アンインストール

パッケージをアンインストールするには **remove** サブコマンドを指定します.

```
rpkg remove パッケージ名 ...
```

アンインストール時には引数で与えられたパッケージの依存関係を解決し, 依存関係を満たす順序でパッケージをアンインストールします.

例: app-A **が** app-B **に依存し,** app-B **が** app-C **に依存している場合**

```
root@NSX:~# rpkg remove app-C app-B app-A
-> app-A をアンインストールする
-> app-B をアンインストールする
-> app-C をアンインストールする
```

C.3 パッケージの一覧

インストール済みパッケージの一覧を表示するには **list** サブコマンドを指定します.

```
rpkg list [ パッケージ名 ... ]
```

パッケージ名はオプションです.

パッケージを指定しない実行例:

```
root@NSX:~# rpkg list
Current:
Rooster-NSX-7000 - 1.5.0 - RoosterOS system file
system-nsx7000 - 1.5.0 - RoosterOS base system
adopt-openjdk-java - 11.0.1.13.1
----
Another:
Rooster-NSX-7000 - 1.5.0 - RoosterOS system file
system-nsx7000 - 1.5.0 - RoosterOS base system
adopt-openjdk-java - 11.0.1.13.1
```

パッケージを指定した実行例:

```
root@NSX:~# rpkg list '*java*'
Current:
adopt-openjdk-java - 11.0.1.13.1
----
Another:
adopt-openjdk-java - 11.0.1.13.1
```

付録 D

Add-on アプリケーションのガイドライン

Add-on アプリケーションのガイドラインは下記のとおりです.

- 将来的に Add-on アプリケーション全体で使用できる RAM の総量を制限する可能性があります. 各 Add-on アプリケーションはこの制限を考慮してください
- 将来的に Add-on アプリケーション全体で使用できる CPU 占有率を制限する可能性があります. 各 Add-on アプリケーションはこの制限を考慮してください
- Add-on アプリケーションがシステムを停止したり, 再起動することを推奨しません
- Add-on アプリケーションがシステム全体に影響する設定を変更することを推奨しません. システム全体に影響する設定の例としては下記のものがあります
  - ルーティング・テーブル設定
  - Netfilter(iptables) 設定
  - sysctl 設定 など
- Add-on アプリケーションがシステムを停止や再起動したり, システム全体に影響する設定変更を行う場合, Add-on アプリケーションのマニュアルなどでその旨を明示し, 利用者に周知してください. 下記についても明示し、周知してください
  - Add-on アプリケーションがシステムを停止するケースが存在する場合, **どのような条件**でシステムを停止するのか
  - Add-on アプリケーションがシステムを再起動するケースが存在する場合, **どのような条件**でシステムを再起動するのか
  - Add-on アプリケーションがシステムの設定を変更するケースが存在する場合, **どの設定項目**に**どのような設定値**を設定するのか
- 最大限に互換性を考慮しますが, システム更新の際に NSX7000 を構成する各コマンドやライブラリを予告なく変更・更新・削除・追加します

  - コマンドの仕様を変更することがあります (例. オプションの削除など)
  - ライブラリの ABI 互換性および API 互換性を失うことがあります

- 厳密に互換性の問題を避けるには下記のいずれかのアプローチを採用してください
  - app.img ファイル内にアプリケーションが依存するすべてのコマンドやライブラリを含め, アプリケーションはそれらを使用する
  - アプリケーションを Java で開発し, 依存する jar ファイルなどを app.img ファイル内に含め, アプリケーションはそれらを使用する
- 一時的なファイルは /app/tmp ディレクトリ以下に, 永続的なファイルは/app/var ディレクトリ以下に作成してください
  - 一時的なファイルは/tmp ディレクトリ以下に作成しても構いませんが, そのサイズは数百 KiB までとしてください. 1MiB を超えるような大きなファイルは/app/var ディレクトリ以下に作成するようにしてください
- フラッシュメモリへの書き込み可能回数を考慮してください. 頻繁かつ大量のログ・メッセージ, /app/var ディレクトリ以下のファイルへの頻繁な書き込みはできるだけ避けるようにアプリケーションを設計してください
- NSX7000 は System V 共有メモリをサポートしません. プロセス間共有メモリが必要な場合, POSIX 共有メモリを使用してください
- NSX7000 は System V セマフォをサポートしません. プロセス間セマフォが必要な場合, POSIX セマフォを使用してください
- NSX7000 は System V メッセージキューをサポートしません. プロセス間メッセージキューが必要な場合, POSIX メッセージキューを使用してください