Synology

Synology DiskStation Manager

3rd-Party Package Developer Guide

Table of Contents

Synology DSM6.0 Developer Guide

Copyright

Getting Started

Synology offers this developer guide with instructions on how to develop and install 3rd-party applications on Synology NAS products or a line of network attached storage devices developed on the Linux kernel. With this guide, you can familiarize yourself with the following procedures:

- Compile programs to run on a Synology NAS.
- Integrate applications with Synology DiskStation Manager (DSM).
- Install application files to the recommended path to keep them intact when DSM is upgraded.
- Integrate applications with the Synology web authentication interface.
- Create a package file for manual or one-click installation in Synology Package Center.

This document is written for Synology users and system integrators who are interested in adding their applications to their Synology NAS. Those who read this document are assumed to have some basic understanding of Linux programming.

System Requirements

Toolkit Requirements

In order to compile programs to run on the Synology NAS, the system must meet the following requirements.

- **64bit** generic linux environment. (For example, Ubuntu 16.04 LTS)
- bash (>= 4.1.5)
- Python (>= 2.7.3)
- Require root permission. (Our toolkit will use `chroot` command)

Please do NOT use Synology NAS base system to install toolkit as your development environment. NAS system is specialized for storage, not for generic developing purpose.

Instead, you can install Synology Docker package on NAS and setup a generic linux container to install the toolkit.

Package Runtime OS Suggestions

The resulting package (.spk) will have best compatibility running on

- Synology NAS version DSM6.0+

Create Package

In this section, we will explain how to create a Synology Package using Package Toolkit.

If you want to build a Synology Package without using Package Toolkit, you must:

- Prepare a cross compile tool chain
- Prepare a build environment
- Prepare metadata
- Compile source code
- Create and sign the package

Creating a package manually can be very complex for most developers, so we recommended using the Package Toolkit to make the package creation process easier. To make the package creation process go smoothly, you will still need to write some scripts describing how you want to build and create your packages.

In the following sub-sections, the necessary scripts will be stated in detail.

You can download the example source from github: Synology OpenSource

Preparation:

In this section, we will guide you through how to set up an environment for building a Synology Package. Detailed steps include:

- Install Toolkit
- Prepare Build Environment
- Prepare GPG Key

Install Toolkit

This tutorial consists of two parts:

- Pre-built environment
- Front-end scripts

Toolkit Installation:

To install the tookit, simply refer to the following steps. First, you need to clone the front-end scripts from this link to your toolkit base. (For DSM 5.x, use this link instead.)

We will use **/toolkit** as toolkit base in this document from now on.

```
mkdir -p /toolkit
git clone {{ book.externalLinks.pkgscripts }} pkgscripts
```

Pre-build Environment:

For faster development, we have prepared several build environments that depend on different architectures for package developers. You don't have to worry about the necessary built-time libraries (.a and .so) and header files(.h and .hpp) when you are developing your package. This is because the build environments already contain some pre-built projects whose executable binaries or shared libraries are built on DSM, for example, **zlib**, **libxml2**, and so on.

Frond-end Scripts:

We have also provided front-end scripts in a folder named "**pkgscripts**" to make the environment deployment, package compilation, and creation of the final package SPK file easier. In most cases, you only need to use these three scripts while developing a package:

- EnvDeploy
- PkgCreate.py
- include/pkg_utils.sh

The next section will guide you through how to establish a build environment and create a Synology Package by using Package Toolkit.

Prepare Build Environment

You can download and set up pre-built environments by using **EnvDeploy** with the following commands. Use `-v` to specify DSM version, `-p` to specify desired platform, and `-t` to specify the file location when you stored the toolkit locally on your development machine. If `-p` is not given, all available platforms for given versions will be set up.

```
cd /toolkit/pkgscripts/
./EnvDeploy -v 6.0 -p x64 # for example
```

The working directory will look like the following figure.The chroot environment to build your own projects will be **ds.${platform}-${version}**. As mentioned before, this toolkit contains some pre-built libraries and headers which can be found under **cross gcc sysroot**. Sysroot is default search path of compiler. If gcc can not find header or library from path user given, gcc will search **sysroot/usr/{lib,include}**.

Available Platforms

You can use one of the following commands to show available platforms. If `-v` is not given, available platforms for all versions will be listed.

```
./EnvDeploy -v 6.0 --list
./EnvDeploy -v 6.0 --info platform
```

Update Environment

Use EnvDeploy again to update your environments. For example, you can update x64 for DSM 6.0 by using the following command.

```
./EnvDeploy -v 6.0 -p x64
```

Remove Environment

To remove a build environment, you need to apply chroot to the build environment. Unmount the **/proc** folder and exit chroot. After that, remove the build environment folder. The following commands illustrate how to remove a build environment with version 6.0 and platform x64.

```
chroot /toolkit/build_env/ds.x64-6.0 umount /proc
rm -rf /toolkit/build_env/ds.x64-6.0
```

Prepare GPG Key

If the build environment is 5.0 or above, the Package Toolkit will use a gpg key to sign the package when creating the spk file. Please refer to Package Signature for more details about package signatures.

If you have your own GPG key (**without a passphrase**) already, you will need to put the private key in **/root/.gnupg** under each platform (/toolkit/build_env/ds.${platform}-6.0/root/.gnupg/).

Generate the GPG key

Requirement: gpg, gpg-agent

```
gpg --gen-key

> Please select what kind of key you want:
   (1) RSA and RSA (default)
> choose key size and enter your name, email
> enter a passphrase: just press Enter without typing any character
```

> **WARNING:** Please make sure that you do not type any characters in the passphrase field, otherwise the build process will **FAIL**.

After completing the steps above, the key will be generated under **~/.gnupg**. Move them with the chroot environment.

```
cp ~/.gnupg/* /toolkit/build_env/ds.${platform}-6.0/root/.gnupg/
```

You can also use the following commands to verify whether the key was successfully imported or not.

```
cd /toolkit/build_env/ds.${platform}-6.0/
chroot .
gpg -K
```

The output may produce the following message.

```
/root/.gnupg/secring.gpg
------------------------
sec   2048R/145E0AFD 2015-12-21
uid                  Synology Inc. <synology_inc@synology.com>
ssb   2048R/E0C20F11 2015-12-21
```

Hello World Package

We use the front-end script **PkgCreate.py** to help us compile source code and pack a **Synology Package** or a **SPK file**.
SPK is the file format used by Synology Package Center to properly install your application. For more details about the structural format of an SPK file, you may refer to Synology Package.

In the following sections, we will guide you on how to create a simple utility program that can print out system memory, and pack it into an SPK file by using **PkgCreate.py**.

For more complicated cases, you can refer to the following examples provided:

- Compile Open Source Project: tmux: If you are interested in porting an open source project to DSM system using tmux or setting up any advanced configurations, you may refer to this section.
- Compile Open Source Project: nmap: If you are interested in porting an open source project to DSM system using nmap or setting up any advanced configurations, you may refer to this section.
- Compile Kernel Module: If you are interested in installing more kernel modules to your DSM system, you may refer to this section.

Create Package Workflow:

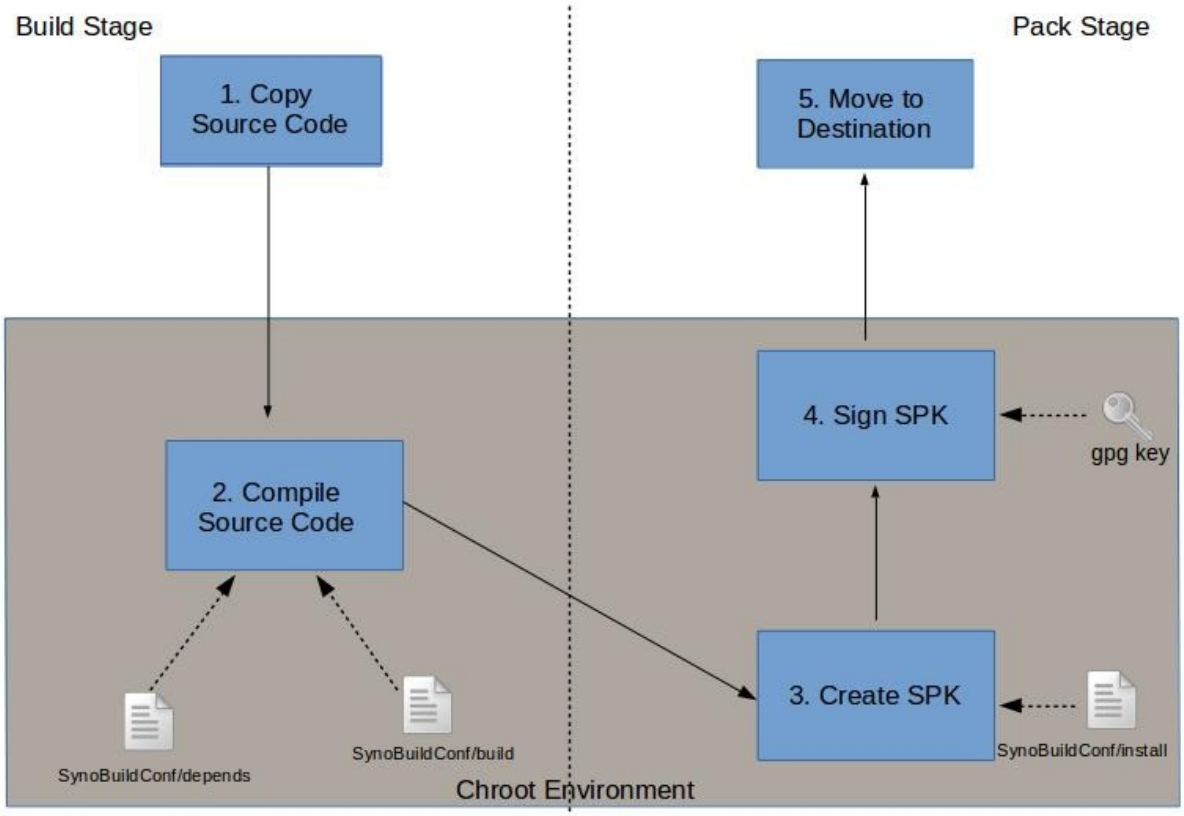
There are two stages in the PkgCreate.py package creation process, the **Build Stage** and the **Pack Stage**.

In the Build Stage, PkgCreate.py will compile your project and all dependent projects in the correct order. In the Pack Stage, PkgCreate.py will pack your project into an SPK file.

To create your SPK file with PkgCreate.py properly, you will need to provide additional configuration files and build scripts to describe how to build your project. These files are put in a folder named "**SynoBuildConf**" under your project. These files and their purpose are listed in below.

- SynoBuildConf/depends: defines the dependency of your project. For further details, please refer to Build Stage
- SynoBuildConf/build: specifies PkgCreate.py on how to compile your project. For further details, please refer to Build Stage
- SynoBuildConf/install: specifies PkgCreate.py on how to pack your SPK file. For further details, please refer to Pack Stage
- SynoBuildConf/install-dev: similar to SynoBuildConf/install, but this will pack your SPK file in chroot environment rather than general DSM system. For further details, please refer to Compile Open Source Project: nmap.

For more details about these configuration files, please refer to Build Stage and Pack Stage. The following figure shows the work flow of these two stages.



You can use the following commands to tell PkgCreate.py how to run through both stages.

```
cd /toolkit
pkgscripts/PkgCreate.py -x0 -c ${project}
```

The `-c` option tells PkgCreate.py to build, pack, and sign your project. The `-x0` option is meant to traverse and build all dependent projects in the correct order. Each project is built according to their own SynoBuildConf/build.

> Note: PkgCreate.py compiles source code and packs your package under **chroot environment**. Therefore, you must run all commands with root permission or with sudo.

PkgCreate.py has many other options to control the build flow. The following subsections will explore deeper into those options or you may directly refer to the Advanced section for more details.

Source Code Layout:

In Build Stage, PkgCreate.py will try to link all the projects to the build environment. As a result, your project source code must be put in a folder (we call it a "**project**") under **/toolkit/source**. The following figure shows the whole working directory as an example.

```
toolkit/
├── build_env/
|   └── ds.${platform}-${version}/
|       └── /usr/syno/
|           ├── bin
|           ├── include
|           └── lib
├── pkgscripts/
└── source/
    └──minimalPkg/
        ├── minimalPkg.c
        ├── INFO.sh
        ├── Makefile
        ├── PACKAGE_ICON.PNG
        ├── PACKAGE_ICON_256.PNG
        ├── scripts/
        |   ├── postinst
        |   ├── postuninst
        |   ├── postupgrade
        |   ├── preinst
        |   ├── preuninst
        |   ├── preupgrade
        |   └── start-stop-status
        └── SynoBuildConf/
            ├── build
            ├── depends
            └── install
```

> Note: You may organize the source code in any structure you like as long as **SynoBuildConf** is edited correctly. The following sections will explain this in detail.

Below is some sample code that we will use as an example.

```
// Copyright (c) 2000-2016 Synology Inc. All rights reserved.

#include <sys/sysinfo.h>
#include <syslog.h>
#include <stdio.h>

int main(int argc, char** argv) {
    struct sysinfo info;
    int ret;

    if (ret != 0) {
        syslog(LOG_SYSLOG, "Failed to get info\n");
        return -1;
    }

    syslog(LOG_SYSLOG, "[MinimalPkg] %s sample package ...", argv[1]);
    syslog(LOG_SYSLOG, "[MinimalPkg] Total Ram: %u\n", (unsigned int)info.totalram);
    syslog(LOG_SYSLOG, "[MinimalPkg] Free RAM: %u\n", (unsigned int)info.freeram);

    return 0;
}
```

Environment Variables in Build and Install Script

Front-end scripts will pass some environment variables to **SynoBuildConf/build** and **SynoBuildConf/install**. You can utilize them to build and install your projects. You can also find most of them in **/toolkit/build_env/ds.${platform}-${version}/{env.mak,env32/64.mak}.** Some of the environment variables are listed in below.

- **CC**: path of gcc cross compiler.
- **CXX**: path of g++ cross compiler.
- **LD**: path of cross compiler linker.

- **CFLAGS**: global cflags includes.
- **AR**: path of cross compiler ar.
- **NM**: path of cross compiler nm.
- **STRIP**: path of cross compiler strip.
- **RANLIB**: path of cross compiler ranlib.
- **OBJDUMP**: path of cross compiler objdump.
- **LDFLAGS**: global ldflags includes.
- **ConfigOpt**: options for configure.
- **ARCH**: processor architecture.
- **SYNO_PLATFORM**: Synology platform.
- **DSM_SHLIB_MAJOR**: major number of DSM (integer).
- **DSM_SHLIB_MINOR**: minor number of DSM (integer).
- **DSM_SHLIB_NUM**: build number of DSM (integer).
- **ToolChainSysRoot**: cross compiler sysroot path.
- **SysRootPrefix**: cross compiler sysroot concat with prefix /usr.
- **SysRootInclude**: cross compiler sysroot concat with include_dir /usr/include.
- **SysRootLib**: cross compiler sysroot concat with lib_dir /usr/lib.

Build Stage:

In the Build Stage, PkgCreate.py will compile the project and its dependent projects. Please note that in this stage, PkgCreate.py depends on two build scripts (**SynoBuildConf/build** and **SynoBuildConf/depend**) to get the necessary information.

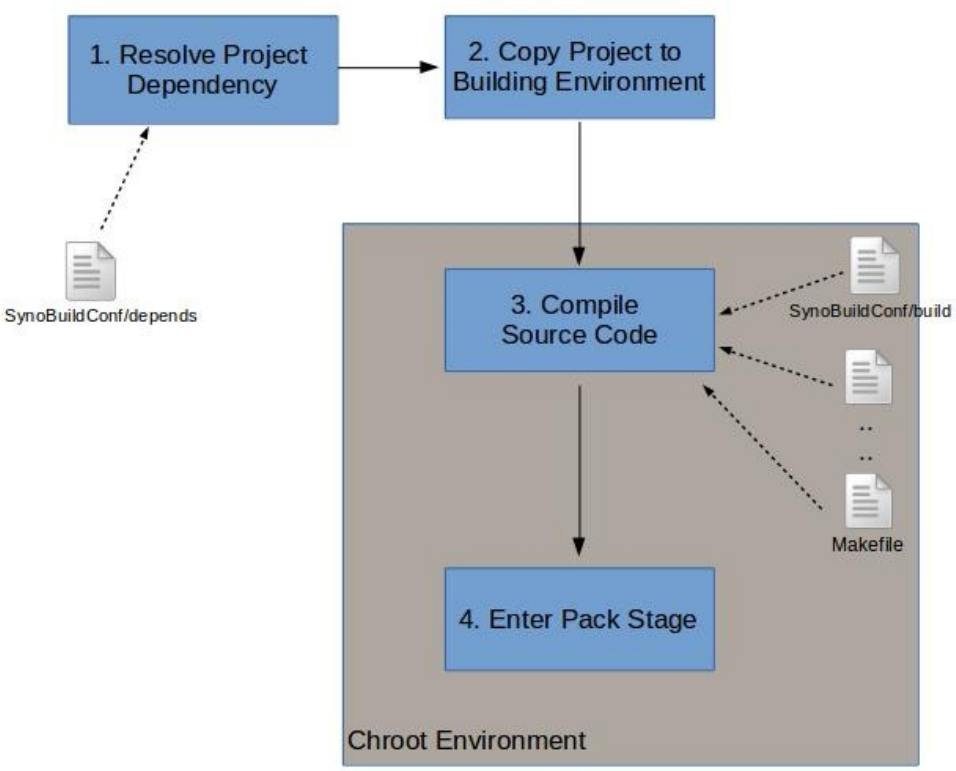
```
PkgCreate.py ${project} # build project
```

Build Stage Workflow:

The workflow of the Build Stage is as follows.

1. Based on your **SynoBuildConf/depend**, PkgCreate.py will locate the target DSM version from [default] section.
2. PkgCreate.py will resolve the projects you depend on.
3. Your project and the dependent projects which are placed under **/toolkit/source** will be hard-linked to **/toolkit/build_env/ds.${platform}/source**.
4. Their **SynoBuildConf/build** will be executed in order according to their dependency based on each **SynoBuildConf/depend**.
5. If your project is needed by other project for cross compiling, you may add **SynoBuildConf/install-dev** script. **install-dev** script will install cross compiled product into platform chroot.

> Note: **SynoBuildConf/build** is executed under chroot environment **/toolkit/build_env/ds.${platform}**.



Build Stage

SynoBuildConf/depends

PkgCreate.py will resolve your dependency according to this configuration file. You need to specify your project dependency and the build environment of your project in this file. For example:

```
[BuildDependent]
# each line here is a dependent project

[ReferenceOnly]
# each line here is a project for reference only but no need to be built

[default]
all="6.0"   # toolkit environment version of specific platform. (all platform use 6.0 toolkit environment)
```

There are three fields in SynoBuildConf/depends.

- **BuildDependent**: Describes other projects which are dependent on this project. For further details about this field, please refer to Compile Open Source Project: nmap.
- **ReferenceOnly**: Describes other projects which are referred by this project, without the build process.
- **default**: Describes the toolkit environment. This section is a necessary field. It indicates each platform to build against some DSM version and the key "**all**" means all platform use this version by default.

You can use **ProjDepends.py** front-end scripts to see whether the dependency order of your projects is correct. Option `-x0` will traverse all dependent projects of **${project}**.

```
cd /toolkit/pkgscripts
./ProjDepends.py -x0 ${project}
```

If your application contains more than one project, put them in **/toolkit/source** and edit **SynoBuildConf** accordingly for each of them.

For more advanced usage of this file, you may refer to Compile Open Source Project and Advanced.

SynoBuildConf/build

SynoBuildConf/build is a shell script that tells PkgCreate.py how to compile your project. The current working directory of this shell script is located in **/source/${project}** under chroot environment.

All pre-built binaries, headers, and libraries are under **cross compiler sysroot** in chroot environment. Since sysroot is the default search path of cross compiler, you do not need to provide `-I` or `-L` to `CFLAGS` or `LDFLAGS`.

Variables:

All variable you can use in `SynoBuildConf/build`:

- **CC**: path of gcc cross compiler.
- **CXX**: path of g++ cross compiler.
- **LD**: path of cross compiler linker.
- **CFLAGS**: global cflags includes.
- **AR**: path of cross compiler ar.
- **NM**: path of cross compiler nm.
- **STRIP**: path of cross compiler strip.
- **RANLIB**: path of cross compiler ranlib.
- **OBJDUMP**: path of cross compiler objdump.
- **LDFLAGS**: global ldflags includes.
- **ConfigOpt**: options for configure.
- **ARCH**: processor architecture.
- **SYNO_PLATFORM**: Synology platform.
- **DSM_SHLIB_MAJOR**: major number of DSM (integer).
- **DSM_SHLIB_MINOR**: minor number of DSM (integer).
- **DSM_SHLIB_NUM**: build number of DSM (integer).
- **ToolChainSysRoot**: cross compiler sysroot path.

- **SysRootPrefix**: cross compiler sysroot concat with prefix /usr.
- **SysRootInclude**: cross compiler sysroot concat with include_dir /usr/include.
- **SysRootLib**: cross compiler sysroot concat with lib_dir /usr/lib.

The example build scripts is like:

```
# SynoBuildConf/build

case ${MakeClean} in
       [Yy][Ee][Ss])
               make distclean
               ;;
esac

make ${MAKE_FLAGS}
```

The above example calls the `make` command and compiles your project according to your **Makefile** located in `/source/${project}` .

Synology toolkit environment has included selected prebuild projects. You can enter the chroot and use following command to check if needed header or project is provided by toolkit.

```
## inner chroot
dpkg -l  # list all dpkg projects.
dpkg -L {project dev} # list project install files
dpkg -S {header/library pattern} # search header/library pattern.
```

For example, the project needs `zlib.h` and `libz.so` in the build stage. Use following command to check if zlib and its component are installed in chroot.

```
chroot /tookit/build_env/ds.x64-6.0/
## inner chroot
>> dpkg -l | grep zlib
ii  zlib-1.x-x64-dev        6.0-7274       all             Synology build-time library

>> dpkg -L zlib-1.x-x64-dev
/.
/usr
/usr/local
/usr/local/x86_64-pc-linux-gnu
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/libz.so
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/libz.a
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/pkgconfig
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/pkgconfig/zlib.pc
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/libz.so.1
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/libz.so.1.2.8
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/include
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/include/zconf.h
/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/include/zlib.h

>> dpkg -S zlib.so
zlib-1.x-x64-dev: /usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/libz.so
zlib-1.x-x64-dev: /usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/libz.so.1.2.8
zlib-1.x-x64-dev: /usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/libz.so.1
```

All file has been installed into sysroot, cross gcc can find zlib.h and libz.so directly.

Some open source require to use other projects' cross compiled product while building their own . For example, `python` needs `libffi` and `zlib` while configure, we need to provide those two project before build `python` . You can install the cross compiled product into the destination you want in build script. Please refer to Compile Open Source Project: nmap for more information.

Makefile

The following example shows a Makefile. Most of the content contains typical makefile rules. Note that when writing your project **Makefile**, you can utilize pre-defined variables in **/env.mak**.

```
# Copyright (c) 2000-2016 Synology Inc. All rights reserved.

## You can use CC CFALGS LD LDFLAGS CXX CXXFLAGS AR RANLIB READELF STRIP after include env.mak
include /env.mak

EXEC= minimalPkg
OBJS= minimalPkg.o

all: $(EXEC)

$(EXEC): $(OBJS)
    $(CC) $(CFLAGS) $< -o $@ $(LDFLAGS)

install: $(EXEC)
    mkdir -p $(DESTDIR)/usr/bin/
    install $< $(DESTDIR)/usr/bin/

clean:
    rm -rf *.o $(EXEC)
```

For more detailed descriptions about makefile, please refer to here.
For a full list of environment variables that are provided by **/env.mak**, please refer to Create Package.

Pack Stage:

In the Pack Stage, PkgCreate.py packs all the necessary files according to your metadata and creates a final Synology Package in the **result_spk** folder. If you want PkgCreate.py to enter the Pack Stage without the Build Stage, simply run **PkgCreate.py** with the `-i` option.

For example, the following command will execute **${project}/SynoBuildConf/install** without signing and putting the final package SPK file (if any) to **/toolkit/result_spk**.

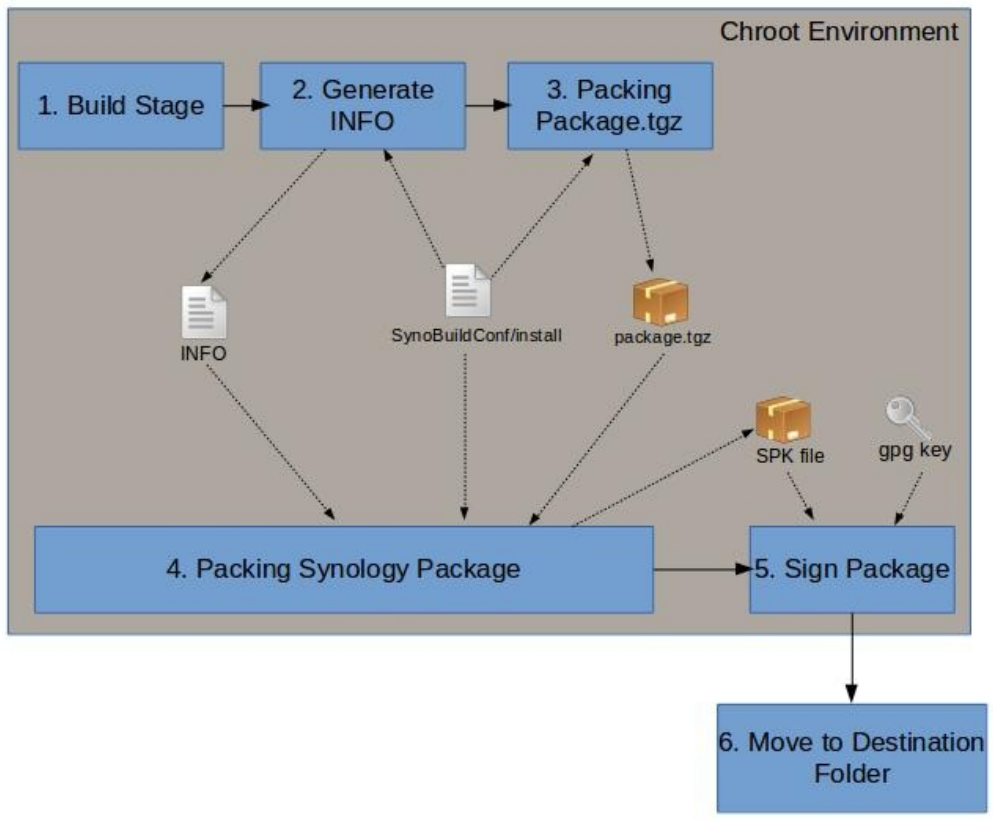
```
cd /toolkit
pkgscripts/PkgCreate.py -i --no-sign ${project}
```

Pack Stage Work Flow:

The workflow of the Pack Stage is as follows.

1. PkgCreate.py will execute the build script **SynoBuildConf/install**.
   i. Create INFO file by using INFO.sh.
   ii. Move necessary files to a temporary folder, **/tmp/_install**, for instance, and create package tgz.
   iii. Move necessary metadata and resources to the temporary folder, **/tmp/_pkg**, for instance, and create the final Synology Package.
2. PkgCreate.py will sign the newly created Synology Package file with a gpg key which is placed under /root/.

The following figure shows the work flow of the Pack Stage.



SynoBuildConf/install

This file must be written in bash and indicates the front-end script on how to pack your project. The current working directory is **/source/${project}** under chroot environment. If this is the top project of your package, this file will define how to create the final package SPK file, including directory structure and the **INFO** file in your SPK. (Please refer to INFO chapter)

Define **SynoBuildConf/install** for installation metadata, resource files and packing your package.

```
#!/bin/bash
# Copyright (C) 2000-2016 Synology Inc. All rights reserved.

### Use PKG_DIR as working directory.
PKG_DIR=/tmp/_test_spk
rm -rf $PKG_DIR
mkdir -p $PKG_DIR

### get spk packing functions
source /pkgscripts/include/pkg_util.sh

create_inner_tarball() {
    local inner_tarball_dir=/tmp/_inner_tarball

    ### clear destination directory
    rm -rf $inner_tarball_dir && mkdir -p $inner_tarball_dir

    ### install needed file into PKG_DIR
    make install DESTDIR="$inner_tarball_dir"

    ### create package.txz: $1=source_dir, $2=dest_dir
    pkg_make_inner_tarball $inner_tarball_dir "${PKG_DIR}"
}

create_spk(){
    local scripts_dir=$PKG_DIR/scripts

    ### Copy Package Center scripts to PKG_DIR
    mkdir -p $scripts_dir
    cp -av scripts/* $scripts_dir

    ### Copy package icon
    cp -av PACKAGE_ICON*.PNG $PKG_DIR

    ### Generate INFO file
    ./INFO.sh > INFO
    cp INFO $PKG_DIR/INFO

    ### Create the final spk.
    # pkg_make_spk <source path> <dest path> <spk file name>
    # Please put the result spk into /image/packages
    # spk name functions: pkg_get_spk_name pkg_get_spk_unified_name pkg_get_spk_family_name
    mkdir -p /image/packages
    pkg_make_spk ${PKG_DIR} "/image/packages" $(pkg_get_spk_family_name)
}

create_inner_tarball
create_spk
```

We will briefly explain the scripts above.

In the beginning, the script will call the **PrepareDirs** function which will prepare the necessary folder for the project.

After creating the folder, the script called **SetupPackageFiles** will move the necessary resource files to **$INST_DIR** and **$PKG_DIR**. In this step, we have called the INFO.sh file to create the **INFO** file. Although you may put the codes that will generate the INFO file in the SynoBuildConf/install script, we highly recommend that you create the **INFO** seperately. Generally, we name it **INFO.sh**. You can see how to write INFO.sh in the following subsections.

- INFO
- scripts
- binary

After moving the resource file to the proper location, we will call the **MakePackage** function to create the package for us. We have included/sourced a helper script called **pkg_util.sh** which is located in /pkgscripts/include. This script creates the final package SPK file in the correct format. The **pkg_make_package** and **pkg_make_spk**, which are defined in **pkg_util.sh*, create the Synology Package. You can see how to use this helper function below.

- **pkg_make_inner_tarball $1 $2**: Create $2/packages.tgz from files in $1.
- **pkg_make_spk $1 $2**: Create $2/spk from files in $1.

INFO.sh

As mentioned earlier, **INFO.sh** is just an optional script. You can create the INFO file by hand or move the code to **SynoBuildConf/install**. However, we strongly recommend that you utilize **INFO.sh** so that you can create the INFO file separately from SynoBuildConf/install.

```
#!/bin/bash
# Copyright (c) 2000-2016 Synology Inc. All rights reserved.

source /pkgscripts/include/pkg_util.sh

package="minimalPkg"
version="1.0.0000"
displayname="Minimal Package"
maintainer="Synology Inc."
arch="$(pkg_get_unified_platform)"
description="this is a minimal package"
[ "$(caller)" != "0 NULL" ] && return 0
pkg_dump_info
```

> Note: The above code is just an example to show some important variables for pkg_dump_info. If you want to know more details about the INFO file and each of the fields, please refer to INFO.

Similar to SynoBuildConf/install, we must first include **pkg_util.sh**. After that, we can set up the proper variables and call the **pkg_dump_info** to create the INFO file correctly.

As you may have noticed, we used another helper function called **pkg_get_platform** to set the arch variable. This variable indicates the current platform we are building.

The following statements indicate how to use the helper functions.

- **pkg_get_spk_platform**: Return platform for “arch” in **INFO**.
- **pkg_dump_info**: Dump **INFO** according to given variables.

After creating the INFO.sh file, remember to **add executable bit** in INFO.sh.

```
chmod +x INFO.sh
```

Pack util functions

Synology package framework provides several functions and and increases efficiency of packing packages. The functions such as generating arch information in the INFO file, separating spk name and creating spk will be enable after executing /pkgscripts/include/pkg_util.sh undefined.

Platform functions

A spk can be installed on one or more platforms. You can decide which platforms can install this spk via spk INFO file.

function name	Values	Description
(No function)	noarch	Package only contain scripts. spk can be run on all synology models.
pkg_get_platform_family (DSM6.0 only)	x86_64 i686 armv7 armv5 ppc...	Unify platforms with same kernel into a `platform family` . spk can run on same family synology models.
pkg_get_spk_platform	bromolow cedarview qoriq armadaxp...	Directly output the platform where the toolkit environment is used. spk only can be run on the specific platform.

- First, if your package doesn't have any binary, you can use **noarch** as the platfrom and write the scripts for your package. Package with `arch=noarch` can be installed onto any synology model.
- Second, if your package doesn't have any kernel related functions, the package can run on the same architecture platforms. Use function `pkg_get_platform_family` to get platform family. Package can be installed on the models which are included in the same platform family. For example, package with `arch=x86_64` can be install onto `bromolow x64 cedarview dockerx64 broadwell` models. Please refer to the platform family map at github. (Note: Platform family only support DSM6.0 or upper version)
- Third, if your package contain kernel related functions, every platforms will need a specific spk. Please use function `pkg_get_spk_platform` to get the platform(s) which be compatiable with your toolkit environment.

spk name functions

After spk generated, we need to distinguish spk name from platform. We can use spk name functions:

Function name	Corresponding platform function	Example	Description
pkg_get_spk_name	pkg_get_spk_platform	minimalPkg-bromolow-1.0.0000.spk / minimalPkg-cedarview-1.0.0000.spk ...	spk name depend on which toolkit environment is using.
pkg_get_spk_name	noarch	minimalPkg-1.0.0000.spk	If package platform="noarch", this function will output spk name without platform info.
pkg_get_spk_family_name	pkg_get_platform_family	minimalPkg-x86_64-1.0.0000.spk	spk name will be unified into platform family. The same platform family will geneate same spk name. i.e bromolow and x64 will has same spk name.

You need to input **INFO path** as argument. If no path input, the function will get INFO file from `$PKG_DIR/INFO` automatically without inputting INFO path.

Create spk function

Developer can use `pkg_make_spk` to create spk.

Usage:

```
pkg_make_spk $source_path $dest_path $spk_name

source_path is spk source directory. All spk files must copy into this direcotry before run pkg_make_spk.
dest_path is target spk path.
spk_name is spk name with/without platform info.

Example:
pkg_make_spk /tmp/_test_spk "/image/packages" $(pkg_get_spk_family_name)
```

Sign Package

In DSM 5.1 and onward, the Package Center has a built-in code sign mechanism to ensure the package's publisher integrity. The toolkit based on DSM 5.0 and onward has the CodeSign.php script to sign the package with GnuPG keys. If you do not have a GPG key, you will need to generate one. Please refer to Prepare GPG Key for more information.

If you want PkgCreate.py to sign the package automatically, you can use the PkgCreate.py without the `--no-sign` option. For example, the following command indicates PkgCreate.py to build and install your project without a signature.

```
PkgCreate.py -i ${project}
```

In addition, if you want to sign the package on your own, you can use the following command to sign your package manually.

```
chroot /toolkit/build_env/ds.${platform}-${version}
php /pkgscripts/CodeSign.php [option] --sign=package-path

Options:
--keydir=keyrings directory (default is /root/.gnupg)
--keyfpr=key's fingerprint (default is "". Under this circumstances, we will using the first key in the key directory t

Examples:
php /pkgscripts/CodeSign.php --sign=phpBB-3.0.12-0031.spk
php /pkgscripts/CodeSign.php --keydir=/root/.gpg --keyfpr=C1BF63CD --sign=phpBB-3.0.12-0031.spk
```

Essential Run Time Files

Scripts:

The "**scripts**" folder contains shell scripts which are executed during the installation, uninstallation, upgrading, starting, and stopping of packages. There are seven script files stored in the "**scripts**" folder:

- postinst
- postuninst
- postupgrade
- preinst
- preuninst
- preupgrade
- start-stop-status

> Note: Even if you do not provide any of these script files, the package will be created without any errors. However, when you try to install your package on your DSM system, Package Center will not be able to install your package.

For simplicity, the scripts will look like the following example.

```
#!/bin/sh
exit 0
```

More details about these scripts will be explained in the scripts section.

Icon:

You can add an icon to your package so that when Package Center shows the information of your package, it will use the icon you have provided instead of the default one.

To add an icon to your package, put the image your want into your project source folder, then slightly modify your SynoBuildConf/install script. The following is an example of a modified SynoBuildConf/install from the previous example. For further details regarding package icons, please refer to the Package Structure section.

The only difference from the previous example is that we copied our image files into the $PKG_DIR and renamed it to **PACKAGE_ICON.PNG** and **PACKAGE_ICON_256.PNG**. inside the **SetupPackageFiles** function.

> Note: Remember to rename your image to **PACKAGE_ICON.PNG** and **PACKAGE_ICON_256.PNG** in all CAPS; otherwise, Package Center will not render the icon properly.

```
#!/bin/bash
# Copyright (C) 2000-2016 Synology Inc. All rights reserved.

### Use PKG_DIR as working directory.
PKG_DIR=/tmp/_test_spk
rm -rf $PKG_DIR
mkdir -p $PKG_DIR

### get spk packing functions
source /pkgscripts/include/pkg_util.sh

create_inner_tarball() {
    local inner_tarball_dir=/tmp/_inner_tarball

    ### clear destination directory
    rm -rf $inner_tarball_dir && mkdir -p $inner_tarball_dir

    ### install needed file into PKG_DIR
    make install DESTDIR="$inner_tarball_dir"

    ### create package.txz: $1=source_dir, $2=dest_dir
    pkg_make_inner_tarball $inner_tarball_dir "${PKG_DIR}"
}

create_spk(){
    local scripts_dir=$PKG_DIR/scripts

    ### Copy Package Center scripts to PKG_DIR
    mkdir -p $scripts_dir
    cp -av scripts/* $scripts_dir

    ### Copy package icon
    cp -av PACKAGE_ICON*.PNG $PKG_DIR

    ### Generate INFO file
    ./INFO.sh > INFO
    cp INFO $PKG_DIR/INFO

    ### Create the final spk.
    # pkg_make_spk <source path> <dest path> <spk file name>
    # Please put the result spk into /image/packages
    # spk name functions: pkg_get_spk_name pkg_get_spk_unified_name pkg_get_spk_family_name
    mkdir -p /image/packages
    pkg_make_spk ${PKG_DIR} "/image/packages" $(pkg_get_spk_family_name)
}

create_inner_tarball
create_spk
```

Summary

In previous sections, we learned how to use Package Tool to compile and pack your project. We will list some important notes here.

Source Code Layout:

```
toolkit/
├── pkgscripts/
├── build_env/
│   └── ds.${platform}-${version}
│       └── /usr/syno
│           ├── lib
│           ├── bin
│           └── include
├── result_spk/
│   └── ${package}-${version}/
│       └── *.spk
└── source/
    └── ${project}/
        ├── source code, Makefile ...
        ├── INFO.sh
        ├── PACKAGE_ICON.PNG
        ├── PACKAGE_ICON_256.PNG
        ├── scripts/
        │   ├── postinst
        │   ├── postuninst
        │   ├── postupgrade
        │   ├── preinst
        │   ├── preuninst
        │   ├── preupgrade
        │   └── start-stop-status
        └── SynoBuildConf/
            ├── build
            ├── depends
            └── install
```

The necessary files required by PkgCreate.py are listed below.

- INFO
- SynoBuildConf/build
- SynoBuildConf/install
- SynoBuildConf/depends
- scripts/postinst
- scripts/postuninst
- scripts/postupgrade
- scripts/preinst
- scripts/preuninst
- scripts/preupgrade
- scripts/start-stop-status

Walk-Through of Create Package

1. Install Package Toolkit
2. List Available Environments
3. Create Building Environment using EnvDeploy
4. Create GPG key
5. Create SynoBuildConf/build SynoBuildConf/install SynoBuildConf/depends

6. Create scripts folder and scripts
7. Build package with PkgCreate.py

Command Walk-Through

```
# Install Package Toolkit
mkdir -p /toolkit
cd /toolkit
git clone https://github.com/SynologyOpenSource/pkgscripts-ng pkgscripts

# Make Build Environment
./pkgscripts/EnvDeploy -v 6.0 --list
./pkgscripts/EnvDeploy -v 6.0 --info platform
./pkgscripts/EnvDeploy -v 6.0 -p x64 # for example

# Prepare GPG key
gpg --gen-key

> Please select what kind of key you want:
   (1) RSA and RSA (default)
> choose key size and enter your name, email
> enter a passphrase: just press Enter without typing any character
cp ~/.gnupg/* /toolkit/build_env/ds.x64-6.0/root/.gnupg/

# Prepare Project folder
mkdir -p ./source/${project}
# add source code, Makefile, SynoBuildConf and scripts
...

# Compile and Build Synology Package
./pkgscripts/PkgCreate.py -x0 -c ${project}
```

Compile Open Source Project

This chapter will show you how to build an open source project for your DSM system using Package Toolkit. If you wish to compile the open source project manually, please refer to Appendix B: Compile Open Source Project Manually.

As mentioned in Create Package, you have to create SynoBuildConf/build, SynoBuildConf/install, and SynoBuildConf/depends before using Package Toolkit.

Unlike the previous example, compiling an application on most open source projects may require executing the following three steps:

1. `configure`
2. `make`
3. `make install`

The configure script consists of many lines which are used to check some details about the machine where the software is going to be installed. This script will also check a lot of dependencies on your system. When you run the configure script, you will see a lot of output on the screen, each being some sort of question with a respective yes/no as a reply. If any of the major requirements are missing on your system, the configure script will exit and you will not be able to proceed with the installation until you meet the required conditions. In most cases, compile applications on some particular target machines will require you to modify the configure script manually to provide the correct values.

When running the configure script to configure software packages for cross-compiling, you will need to specify the `CC`, `LD`, `RANLIB`, `CFLAGS`, `LDFLAGS`, `host`, `target`, and `build`.

In this chapter, we will use **platform x64** as our example.

Preparation:

First download the tmux source code from the official github site or you can download tmux from this link.

> Note: The archive file you've downloaded from the above links is different from the official tmux source code. We have added the necessary build scripts.

Project Layout:

```
tmux/
    ├── tmux related source code
    ├── INFO.sh
    ├── scripts/
    └── SynoBuildConf/
        ├── build
        ├── depends
        └── install
```

SynoBuildConf/depends:

The following is the **depends** file for this example. There is nothing special about the **depends** file.

```
[default]
all="6.0"
```

SynoBuildConf/build:

The build script is slightly different from the previous one. Here you will have to pass the following environment variables to configure:

- CC
- AR
- CFLAGS
- LDFLAGS

In addition, since tmux is dependent on ncurses, you will need to use `pkg-config` to resolve the necessary header files and libraries for tmux.

The following is an example of SynoBuildConf/build:

```
#!/bin/sh
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.

include /env.mak

case ${MakeClean} in
    [Yy][Ee][Ss])
        make distclean
        ;;
esac

NCURSES_INCS="`pkg-config ncurses --cflags`"
NCURSES_LIBS="`pkg-config ncurses --libs`"

CFLAGS+="${CFLAGS} ${NCURSES_INCS}"
LDFLAGS+="${LDFLAGS} ${NCURSES_LIBS}"

env CC="${CC}" AR=${AR} CFLAGS="${CFLAGS}" LDFLAGS="${LDFLAGS}" \
./configure ${ConfigOpt}

make ${MAKE_FLAGS}
```

SynoBuildConf/install

Instead of copying the binary to the destination folder, most big projects will use `make install` to install the binaries and libraries. You can pass the DESTDIR environment variable to specify where you want to install the binaries and libraries.

```
#!/bin/bash
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.

PKG_NAME="tmux"
INST_DIR="/tmp/_${PKG_NAME}"
PKG_DIR="/tmp/_${PKG_NAME}_pkg"
PKG_DEST="/image/packages"

PrepareDirs() {
    for dir in $INST_DIR $PKG_DIR; do
        rm -rf "$dir"
    done
    for dir in $INST_DIR $PKG_DIR $PKG_DEST; do
        mkdir -p "$dir"
    done
}

SetupPackageFiles() {
    DESTDIR="${INST_DIR}" make install

    ./INFO.sh > INFO
    cp INFO "${PKG_DIR}"
    cp -r scripts/ "${PKG_DIR}"
}

MakePackage() {
    source /pkgscripts/include/pkg_util.sh
    pkg_make_package $INST_DIR $PKG_DIR
    pkg_make_spk $PKG_DIR $PKG_DEST
}

main() {
    PrepareDirs
    SetupPackageFiles
    MakePackage
}

main "$@"
```

INFO.sh

As mentioned before, we will use INFO.sh to generate the INFO file.

```
#!/bin/sh
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.

. /pkgscripts/include/pkg_util.sh
package="tmux"
version="1.9-a"
displayname="tmux"
arch="$(pkg_get_platform) "
maintainer="Synology Inc."
description="This package will install tmux in your DSM system."
[ "$(caller)" != "0 NULL" ] && return 0
pkg_dump_info
```

> Note: Remember to set the executable bit of INFO.sh file.

Build and Create Package:

Run the following commands to compile the source code and build the package.

```
/toolkit/pkgscripts/PkgCreate.py -p x64 -c tmux
```

After the build process, you can check the result in `/toolkit/result_spk` .

Verify the Result

If the building process was successful, you will see that the .spk file has been placed under result_spk folder. To test the spk file, You can use manual install in Package Center to install your package.

> **Warning**: Remember to import your keys to the DSM system or select **Any publisher** in Package Center->Settings->General->Trust Level. Otherwise, the installation will fail.

You can then try to connect to the DSM using ssh and type the following command to fully scan your DSM machine.

```
cd /var/packages/"${PKG_NAME}"/target/usr/local/bin
./tmux
```

Compile Open Source Project: nmap

This chapter will show you how to build an open source project for your DSM system using Package Toolkit.
The open source project that we are going to build in this example is **nmap**, a network scanning program. We will use **x64** as our build environment platform.

If you wish to compile an open source project manually, please refer to Appendix B: Compile Open Source Project Manually.

As we have mentioned in Hello World Package, you have to create the SynoBuildConf/build, SynoBuildConf/install, and SynoBuildConf/depends before using Package Toolkit.

Unlike the previous example, compiling an application on most open source projects may require executing the following three steps:

1. `configure`
2. `make`
3. `make install`

The configure script consists of many lines which are used to check some details about the machine where the software is going to be installed. This script will also check a lot of dependencies on your system. When you run the configure script, you will see a lot of output on the screen, each being some sort of question with a respective yes/no as a reply. If any of the major requirements are missing on your system, the configure script will exit and you will not be able to proceed with the installation until you meet the required conditions. In most cases, compile applications on some particular target machines will require you to modify the configure script manually to provide the correct values.

When running the configure script to configure software packages for cross-compiling, you will need to specify the `CC`, `LD`, `RANLIB`, `CFLAGS`, `LDFLAGS`, `host`, `target`, and `build`.

Preparation:

First, you will need to download the nmap source code from the official github site. You will also need to download the libpcap source code since nmap depends on libpcap. The libpcap source code can be found here.

The following commands will download the source code for libpacp and nmap.

```
wget https://nmap.org/dist/nmap-7.01.tar.bz2
tar xvf nmap-7.01.tar.bz2 -C /toolkit/source
mv /toolkit/source/nmap-7.01 /toolkit/source/nmap

wget http://www.tcpdump.org/release/libpcap-1.6.2.tar.gz
tar xvf libpcap-1.6.2.tar.gz -C /toolkit/source
mv /toolkit/source/libpcap-1.6.2 /toolkit/source/libpcap
```

Or use git to download source code

```
cd /toolkit/source
git clone https://github.com/nmap/nmap

git clone https://github.com/the-tcpdump-group/libpcap
cd libpcap
git checkout origin/libpcap-1.6
```

Please remember to upgrade the libpacp to version 1.6 or the build package process will fail.

Project Layout:

After you download the source code, your toolkit layout should look like the following figure.

```
toolkit/
├── build_env/
│   └── ds.${platform}-${version}/
│       └── /usr/syno/
│           ├── bin
│           ├── include
│           └── lib
├── pkgscripts/
└── source/
    ├──nmap/
    │   ├── nmap related source code
    │   ├── INFO.sh
    │   ├── Makefile
    │   ├── Synoscripts/ # nmap has it's own scripts folder
    │   └── SynoBuildConf/
    │       ├── build
    │       ├── depends
    │       └── install
    └──libpcap/
        ├── libpcap related source code
        ├── Makefile
        └── SynoBuildConf/
            ├── build
            ├── depends
            ├── install-dev
            └── install
```

The file, **install-dev**, is a special file which we will be covered in the following section.

SynoBuildConf/depends:

The SynoBuildConf/depends for nmap is slightly different from the previous example. Since nmap depends on libpcap, we have to add the value to the BuildDependent field, so that the PkgCreate.py can resolve the dependency and compile the project in the correct order.

The **depends** file for nmap is as follows.

```
[BuildDependent]
libpcap

[default]
all="6.0"
```

However, the SynoBuildConf/depends for libpcap is the same as the Hello World Example.

```
[BuildDependent]

[default]
all="6.0"
```

SynoBuildConf/build:

The SynoBuildConf/build script is also different from the previous one.

Here you will have to pass several environment variables to configure, so that nmap can be compiled properly

- CC
- CXX
- LD
- AR
- STRIP
- RANLIB

- NM
- CFLAGS
- CXXFLAGS
- LDFLAGS

Since nmap will be compiled with many features by default, we will need to disable some of them to make it clean. The following list contains the features that will be disabled:

- ndiff
- zenmap
- nping
- ncat
- nmap-update
- liblua

> Note: If you are interested in some of the above features and you want to enable them, just change the `--without-${feature}` into `--with-${feature}`.

The following is the SynoBuildConf/build for nmap

```
#!/bin/sh
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.

include /env.mak

PKG_NAME=nmap
INST_DIR=/tmp/_${PKG_NAME}

case ${MakeClean} in
    [Yy][Ee][Ss])
        make distclean
        ;;
esac

env CC="${CC}" CXX="${CXX}" LD="${LD}" AR=${AR} STRIP=${STRIP} RANLIB=${RANLIB} NM=${NM} \
    CFLAGS="${CFLAGS}" CXXFLAGS="$CXXFLAGS $CFLAGS" \
    LDFLAGS="${LDFLAGS} -lnl -lnl-genl -ldbus-1" \
    ./configure ${ConfigOpt} \
    --prefix=${INST_DIR} \
    --without-ndiff \
    --without-zenmap \
    --without-nping \
    --without-ncat \
    --without-nmap-update \
    --without-liblua \
    --with-libpcap=/usr/local

make ${MAKE_FLAGS}
```

In this example, `--with-libpcap` is assigned with value `/usr/local`. We need to install libpcap's cross compiled product into "/usr/local" so that nmap's configure can retrieve libpcap correctly.

The following is the SynoBuildConf/build for libpcap.

```
#!/bin/bash
# Copyright (c) 2000-2012 Synology Inc. All rights reserved.

case ${MakeClean} in
    [Yy][Ee][Ss])
        make distclean
        ;;
esac

case ${CleanOnly} in
    [Yy][Ee][Ss])
        return
        ;;
esac

# prefix with /usr/local, all files will be installed into /usr/local
env CC="${CC}" CXX="${CXX}" LD="${LD}" AR=${AR} STRIP=${STRIP} RANLIB=${RANLIB} NM=${NM} \
    CFLAGS="${CFLAGS} -Os" CXXFLAGS="${CXXFLAGS}" LDFLAGS="${LDFLAGS}" \
    ./configure ${ConfigOpt} \
    --with-pcap=linux --prefix=/usr/local

make ${MAKE_FLAGS}

make install
```

The above script will install libpcap related files into `/usr/local/` in chroot environment. After installing libpcap, nmap can find libpcap's cross compiled products in `/usr/local`.

Synology toolkit provides `libpcap` in chroot.

```
> dpkg -l | grep libpcap
ii  libpcap-x64-dev                 6.0-7274     all        Synology build-time library
```

nmap can use chroot's libpcap by using `${SysRootPrefix}` variable.

```
--with-libpcap=${SysRootPrefix}
```

SynoBuildConf/install

Instead of copying the binary to the destination folder, most big projects will use `make install` to install the binaries and libraries. Since we have used the `--prefix` flag when configuring the nmap project, we can just execute **make install** and it will install the nmap related files to the folder specified by `--prefix`.

```
#!/bin/bash
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.

PKG_NAME="nmap"
INST_DIR="/tmp/_${PKG_NAME}"
PKG_DIR="/tmp/_${PKG_NAME}_pkg"
PKG_DEST="/image/packages"

PrepareDirs() {
    for dir in $INST_DIR $PKG_DIR; do
        rm -rf "$dir"
    done
    for dir in $INST_DIR $PKG_DIR $PKG_DEST; do
        mkdir -p "$dir"
    done
}

SetupPackageFiles() {
    make install
    ./INFO.sh > INFO
    cp INFO "${PKG_DIR}"
    cp -r scripts/ "${PKG_DIR}"
}

MakePackage() {
    source /pkgscripts/include/pkg_util.sh
    pkg_make_package $INST_DIR $PKG_DIR
    pkg_make_spk $PKG_DIR $PKG_DEST
}

main() {
    PrepareDirs
    SetupPackageFiles
    MakePackage
}

main "$@"
```

INFO.sh

As mentioned before, we will use INFO.sh to generate the INFO file.

```
#!/bin/sh
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.

. /pkgscripts/include/pkg_util.sh
package="nmap"
version="7.01"
displayname="nmap"
arch="$(pkg_get_platform) "
maintainer="Synology Inc."
description="This package will install nmap in your DSM system."
[ "$(caller)" != "0 NULL" ] && return 0
pkg_dump_info
```

> Note: Remember to set the executable bit of INFO.sh file.

Build and Create Package:

Lastly, run the following commands to compile the source code and build the package.

```
/toolkit/pkgscripts/PkgCreate.py -p x64 -x0 -c nmap
```

After the build process, you can check the result in `/toolkit/result_spk` .

Verify the Result

If the packing process was successful, you will see an spk file placed in the result_spk folder. To test the spk file, you can use manual install in DSM Package Center to install your package.

> **Warning**: Remember to import your keys to the DSM system or select **Any publisher** in Package Center->Settings->General->Trust Level. Otherwise, the installation will fail.

You can then try to connect to the DSM using ssh and type the following command to fully scan your DSM machine.

```
cd /var/packages/nmap/target/usr/local/bin
./nmap -v -A localhost
```

Compile Kernel Modules

In this chapter, we will provide the tutorial of building Linux kernel modules for the DSM system with Package Toolkit.

If you have the requirement of compiling kernel modules manually, please refer to Appendix B: Compile Kernel Modules Manually.

Use toolkit to build kernel modules

Synology toolkit contain kernel-dev which pack all kernel developer components in. You can follow the directions in pkgscripts-ng to set the environment up.

Kernel related variables

pkgscripts-ng build framework provides several variables which are different between platforms.

- CROSS_COMPILE: Cross-compiler toolchain prefix.
- ARCH: Target architecture.
- KSRC: The path of the store kernel source cross-compiled product. It contains components for compiling the kernel modules.

Example:

- In bromolow:

```
ROSS_COMPILE=/usr/local/x86_64-pc-linux-gnu/bin/x86_64-pc-linux-gnu-
ARCH=x86_64
KSRC=/usr/local/x86_64-pc-linux-gnu/x86_64-pc-linux-gnu/sys-root/usr/lib/modules/DSM-6.1/build
```

- In comcerto2k:

```
CROSS_COMPILE=/usr/local/arm-unknown-linux-gnueabi/bin/arm-unknown-linux-gnueabi-
ARCH=arm
KSRC=/usr/local/arm-unknown-linux-gnueabi/arm-unknown-linux-gnueabi/sysroot/usr/lib/modules/DSM-6.1/build
```

Usage

```
make KSRC="$KSRC" CROSS_COMPILE="$CROSS_COMPILE" ARCH="$ARCH"
```

Sample package: HelloKernel

HelloKernel is a sample package for building kernel module. You can get HelloKernel at github.

Before starting

Please set up your build environment by following pkgscripts-ng.

Getting started

First, preparing a simple kernel module which prints message after `insmod` / `rmmod` the module.

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

int init_module(void) {
    printk(KERN_INFO "Synology HelloKernel package is installed.\n");
    return 0;
}

void cleanup_module(void) {
    printk(KERN_INFO "Synology HelloKernel package has been removed.\n");
}
```

Then, creating the Makefile and using KSRC variable to assign kernel source directory.

```
HELLO_KERNEL= hello_kernel.ko

all: $(HELLO_KERNEL)

obj-m := hello_kernel.o

$(HELLO_KERNEL):
    make -C $(KSRC) M=$(PWD) modules

install: $(HELLO_KERNEL)
    mkdir -p $(DESTDIR)/hello_kernel/
    install $< $(DESTDIR)/hello_kernel/

clean:
    rm -rf *.o $(HELLO_KERNEL) *.cmd
```

In order to be compatible with the framework of Synology packages, you need to create the build script(SynoBuildConf/build) and assign kernel-dev variables to Makefile for compiling kernel modules.

```
#!/bin/bash
# Copyright (C) 2000-2017 Synology Inc. All rights reserved.

case ${MakeClean} in
    [Yy][Ee][Ss])
        make clean
        ;;
esac

make ${MAKE_FLAGS} KSRC=$KSRC CROSS_COMPILE=$CROSS_COMPILE ARCH=$ARCH
```

In addition, you can use `PkgCreate.py` to verify the build script. Please refer to the example of armadaxp as follows:

```
pkgscripts-ng/PkgCreate.py -p armadaxp -I HelloKernel
```

After running `PkgCreate.py`, `hello_kernel.ko` will genaratedi (File path: build_env/ds.armadaxp-6.1/source/HelloKernel/hello_kernel.ko). You can run `insmod hello_kernel.ko` on armadaxp series model such as ds214+...etc. If kernel module insert successfully, the message `"Hi~ Synology kernel package installed."` will show in `dmesg`.

Synology provide the example programs on github[https://github.com/SynologyOpenSource/HelloKernel]. You can clone the HelloKernel repo and use `PkgCreate.py` to generate spk.

```
pkgscripts-ng/PkgCreate.py -c HelloKernel
```

Advanced

This section illustrates advanced types of usage for the Package Toolkit.

PkgCreate.py Command Option List

The following table lists some of the PkgCreate.py commands.

Option Name	Option Purpose
(default)	Run build stage only which include link and compile source code. It's the same as -U option.
-p	Specify the platform you want to pack your project.
-x	Build dependent project level.
-c	Run both **build stage** and **pack stage** which include link source code, compile source code, pack package and sign the final spk.
-U	Run build stage only which includes link and compile source code.
-l	Run build stage only, but will only link your source code.
-L	Run build stage only, but will compile your source code only.
-I	Run pack stage only, which will pack and sign your spk.
--no-sign	Tells PkgCreat.py not to sign your spk file. for example, PkgCreat.py -I --no-sign ${project}
-z	Run all platforms concurrently.
-J	Compile your project with -J make command options.
-S	Disable silent make.

The following table shows the relationship between command options in different stages. You can choose the proper options based on your needs. Option `-c` is enough for most cases.

Stage	Action	(default)	-l	-L	-U	-I --no-sign	-I	-c
Build Stage	Link Source code	Yes	Yes	No	Yes	No	No	Yes
Build Stage	Compile Source code	Yes	No	Yes	Yes	No	No	Yes
Pack Stage	Pack Package	No	No	No	No	Yes	Yes	Yes
Pack Stage	Sign Package	No	No	No	No	No	Yes	Yes

Platform-Specific Dependency

Platform-specific dependency means you can have several dependent projects for different platforms by appending "**:${platform}**" to the following sections: **BuildDependent** and **ReferenceOnly**. The following example shows 816x and aramda370 projects that are on libbar-1.0.

```
# SynoBuildConf/depends

[BuildDependent]
libfoo-1.0

[BuildDependent:816x,armada370]
libfoo-1.0
libbar-1.0

[default]
all="6.0"
```

Collect the SPK File in Your Own Way

By default, **PkgCreate.py** will move the SPK file to **/toolkit/result_spk** according to **/toolkit/build_env/ds.${platform}-${version}/source/${project}/INFO.** You can have your own collect operation by adding a hook, **SynoBuildConf/collect**. **SynoBuildConf/collect** can be any executable shell script (so remember to chmod +x) and **PkgCreate.py** will pass the following environment variables to it:

- **SPK_SRC_DIR**: Source folder of target SPK file.
- **SPK_DST_DIR**: Default destination folder to put SPK file.
- **SPK_VERSION**: Version of package (according to INFO).

The current working directory of **SynoBuildConf/collect** is **/source/${project}** will be under chroot environment.

Package Introduction

In package, you defines some scripts and metadata to control the installation, un-installation, upgrading, starting and stopping processes as well as how to communicate with Synology Package Center in DSM. Synology Package Center provides user interface to the end user and automates the processes and configuring packages to make the end user install, un-install, upgrade, start and stop your package easily.

Package Structure

A Synology package is a SPK file in tar format, containing metadata and files as in the following:

File/Folder Name	Description	File/Folder Type	DSM Requirement
INFO	This file contains the information displayed in Package Center or to control the flow of installation. (Please refer to INFO section for more information)	File	2.0-0731
WIZARD_UIFILES	Optional. This folder contains files where descriptions of UI components are shown during the installation, un-installation, and upgrading process. (Please refer to WIZARD_UIFILES section for more information)	Folder (Contains install_uifile, upgrade_uifile, uninstall_uifile, ...)	3.2-1922
package.tgz	This is a compressed file, containing all the files that are required, such as executable binary, library, or UI files. (Please refer to package.tgz section for more information)	.tgz File	2.0-0731
scripts	This folder contains shell scripts which are executed during the installation, uninstalling, upgrading, starting, and stopping processes. (Please see the scripts section for more information)	Folder (Contains preinst, postinst, preuninst, postunist, preupgrade, postupgrade, start-stop-status)	2.0-0731
conf	Optional. This folder contains configurations. Note: 1. In DSM 4.2 ~ DSM 5.2, if you want to configure files within it, the support_conf_folder key in the INFO file must be set to "yes". 2. In DSM 6.0, you don't need to define the support_conf_folder key in the INFO file. (Please refer to conf section for more information)	Folder (contains PKG_DEPS, PKG_CONX, ...)	4.2-3160
LICENSE	Optional. This file is shown in the installation process, and must be less than 1 MB.	File	3.2-1922
PACKAGE_ICON.PNG	72 x 72 .png image is shown in Package Center	.png file	3.2-1922
PACKAGE_ICON_120.PNG (Deprecated)	120 x 120 .png image is shown in Package Center. Note: It is not compatible with all DSM versions because the icon will not be installed in DSM 4.1 or older. If your package can be installed in DSM 4.1 or older, please refer to the next section to define package_icon_120 in the INFO file instead of taking PACKAGE_ICON_120.PNG.	.png file	4.2-3160 ~ 4.3-3810
PACKAGE_ICON_256.PNG	256 x 256 .png image is shown in Package Center. Note: It is not compatible with all DSM versions because the icon will not be installed in DSM 4.3 or older. If your package can be installed in DSM 4.3 or older, please refer next section to define package_icon_256 in INFO file to instead of taking PACKAGE_ICON_256.PNG.	.png file	5.0-4400

Note:

- All words are case sensitive.
- You can use **PkgCreate.py** and **pkg_make_spk** to help you create the package. For more details, please refer to
  Build and Create Package
  Pack Stage: SynoBuildConf/install

INFO

The "**INFO**" file is used to describe the information of the package. Package Center will search for information on how to control the installation, un-installation, upgrading, starting and stopping processes and listings in Package Center. For example, if you would like the package to be dependent on some services, you can define the key as **install_dep_services**. If you would like to restart some services after the installation process, you can define the key as **instuninst_restart_services**. Package Center will put the "**INFO**" file to **/var/packages/[package identify]/INFO** after the package is installed.

INFO Field Format:

Each piece of information in the **INFO** file is defined by key/value pairs separated by an equals sign
e.g. **key="value"**.

> Note: All words in key and value are case sensitive.

INFO Field List:

There are many fields in an INFO file. We can divide them into two groups:

- Necessary fields: Necessary Fields
- Optional fields: Optional Fields

The following are some Code Words for the INFO field list:

- **apache-sys** = apache daemon listening on DSM ports (e.g. 5000 or 5001)
- **apache-web** = apache daemon listening on Web Station ports (e.g. 80 or 443).
- **mdns** = Multicast DNS Service Discovery
- **db** = MySQL and PostgreSQL
- **apple** network = Apple Network
- **nfs** = NFS
- **ssh** = SSH, Secure Shell
- **pgsql** = PostgreSQL

The version of DSM requirement means key/value pairs in **INFO** works correctly in the minimum version of DSM.

Writing INFO File:

Instead of writing the INFO file by yourself, we have provided useful helper functions in Package Toolkit that will help you create the INFO file. Please refer to Pack Stage: INFO.sh.

Field Name: package

- **Description:** Package identity. No more than one version of a package can exist at the same time in the end user's DSM; therefore, the identification is unique to identify your package. Besides, Package Center will create a /var/packages/[package identity] folder to put package files.
  > Note: This value of the key cannot contain any of these special characters :, /, >, <, | or =.
- **Value:** String
- **Default Value:** (Empty)
- **Example:**

```
package="DownloadStation"
```

- **DSM Requirement:** 2.0-0731

Field Name: displayname

- **Description:** Package Center shows the name of the package.
  > Note: If **displayname** key is empty, Package Center will display the value of **package** key.
- **Value:** String
- **Default Value:** The value of **package** key
- **Example:** None
- **DSM Requirement:** 2.3-1118

Field Name: version

- **Description:** Package version. End users can identify the package version.

  > Note:
  >
  > 1. Each version delimiter is only allowed to be . - or _.
  > 2. Each version number which is delimited by delimiteris only allowed to be number
- **Value:** String
- **Default Value:** (Empty)
- **Example:**

```
version="3.6-3263"
```

- **DSM Requirement:** 2.0-0731

Field Name: firmware

- **Description:** Earliest version of DSM firmware that is required to run the package.
  > Note: Deprecated after 6.1-14715, use *os_min_ver* instead.
- **Value:** X.Y-Z DSM major number, DSM minor number, DSM build number
- **Default Value:** (Empty)
- **Example:** None
- **DSM Requirement:** 2.3-1118

Field Name: os_min_ver

- **Description:** Earliest version of DSM that is required to run the package.
- **Value:** X.Y-Z DSM major number, DSM minor number, DSM build number
- **Default Value:** None
- **Example:**

```
os_min_ver="6.1-14715"
```

- **DSM Requirement:** 6.1-14715

Field Name: description

- **Description:** Package Center shows a short description of the package.
- **Value:** String
- **Default Value:** (Empty)
- **Example:**

```
description = "Download Station is a web-based download application which allows you to download files from the Int
```

- **DSM Requirement:** 2.3-1118
- **DSM Requirement:** 4.2-3160

Field Name: arch

- **Description:** List the CPU architectures which can be used to install the package.
- **Value:** (arch values are separated with a space. Please refer Appendix A: Platform and Arch Value Mapping Table to more information)
- **Default Value:** noarch

> Note:
>
> 1. "noarch" means the package can be installed and work in any platform. For example, the package is written in PHP or shell script.
> 2. Please not pack all binary files with different platforms to one package spk file.

- **Example:**

```
arch="noarch"
or
arch="x86 alpine".
```

- **DSM Requirement:** 2.0-0731

Field Name: maintainer

- **Description:** Package Center shows the developer of the package.
- **Value:** String

- **Default Value:** (Empty)
- **Example:**

```
maintainer="Synology Inc."
```

- **DSM Requirement:** 2.0-0731

Field Name: package_icon

- **Description:** 72x72 png image data is encoded by Base64.

> Note:
>
> 1. This value will be replaced when a **PACKAGE_ICON.PNG** file is stored in the **[package name].spk**.
> 2. If the value is not defined and no **PACKAGE_ICON.PNG** file is in the **[package name].spk**, the package icon will be the default one.

- **Value:** Base64-encoded value
- **Default Value:** a default icon
- **Example:** None
- DSM Requirement: 3.2-1922

Field Name: package_icon_256

- **Description:** 256x256 png image data is encoded by Base64.

> Note:
>
> 1. This value will be replaced when a **PACKAGE_ICON_256.PNG** file is stored in the **[package name].spk**.
> 2. If the value is not defined and no **PACKAGE_ICON_256.PNG** file is in the **[package name].spk**, a 72x72 icon will be the default, and the results will look more pixelated and blurry in DSM.

- **Value:** Base64-encoded value
- **Default Value:** a default icon
- **Example:** None
- **DSM Requirement:** 5.0-4458

Field Name: displayname_[DSM language]

- **Description**: Package Center shows the name in the DSM language set by the end-user. DSM supports the following languages:
  - enu (English)
  - cht (Traditional Chinese)
  - chs (Simplified Chinese)
  - krn (Korean)
  - ger (German)
  - fre (French)
  - ita (Italian)
  - spn (Spanish)
  - jpn (Japanese)
  - dan (Danish)
  - nor (Norwegian)
  - sve (Swedish)
  - nld (Dutch)
  - rus (Russian)
  - plk (Polish)
  - ptb (Brazilian Portuguese)
  - ptg (European Portuguese)
  - hun (Hungarian)
  - trk (Turkish)
  - csy (Czech)
- **Value**: String
- **Default Value**: package name
- **Example**:

```
displayname_enu="Hello World"
displayname_cht=""
```

- **DSM Requirement**: 2.3-1118

Field Name: description_[DSM language]

- **Description**: Package Center shows a short description in the DSM language set by the end-user.
  DSM supports the following languages:
  - enu (English)
  - cht (Traditional Chinese)
  - chs (Simplified Chinese)
  - krn (Korean)
  - ger (German)
  - fre (French)
  - ita (Italian)
  - spn (Spanish)
  - jpn (Japanese)
  - dan (Danish)
  - nor (Norwegian)
  - sve (Swedish)
  - nld (Dutch)
  - rus (Russian)

  - plk (Polish)
  - ptb (Brazilian Portuguese)
  - ptg (European Portuguese)
  - hun (Hungarian)
  - trk (Turkish)
  - csy (Czech)
- **Value**: String
- **Default Value**: description
- **Example**:

```
description_enu="Hello World"
description_cht=""
```

- **DSM Requirement**: 2.3-1118

Field Name: maintainer_url

- **Description**: If a package has a developer webpage, Package Center will show a link to let the user open it.
- **Value**: String
- **Default Value**: (Empty)
- **Example**:

```
maintainer_url="http://www.synology.com"
```

- **DSM Requirement**: 4.2-3160

Field Name: distributor

- **Description**: Package Center shows the publisher of the package.
- **Value**: String
- **Default Value**: (Empty)
- **Example**:

```
distributor="Synology Inc."
```

- **DSM Requirement**: 4.2-3160

Field Name: distributor_url

- **Description**: If a package is installed and has a distributer webpage, Package Center will show a link to let the user open it.
- **Value**: String
- **Default Value**: (Empty)
- **Example**:

```
distributor_url ="http://www.synology.com/enu/apps/3rd-party_application_integration.php"
```

- **DSM Requirement**: 4.2-3160

Field Name: support_url

- **Description**: Package Center shows a support link to allow users to seek technical support when needed.

- **Value**: String
- **Default Value**: (Empty)
- **Example**:

```
support_url="https://myds.synology.com/support/support_form.php".
```

Field Name: support_center

- **Description**: If set to “yes,” Package Center displays a link to make the end user launch Synology Support Center Application when your package is installed.

  > Note: If set to “yes,” the **report_url** link won’t show in Package Center.

- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**: None
- **DSM Requirement**: 5.0-4458

Field Name: model

- **Description**: List of models on which packages can be installed in spesific models. It is organized by Synology string, architecture and model name.
- **Value**: (models are separated with a space, e.g. synology_88f6281_209, synology_cedarview_rs812rp+, synology_x86_411+II, synology_bromolow_3612xs, synology_cedarview_rs812rp+, …)
- **Default Value**: (Empty)
- **Example**:

```
model="synology_bromolow_3612xs synology_cedarview_rs812rp+".
```

- **DSM Requirement**: 4.0-2219

Field Name: exclude_arch

- **Description**: List the CPU architectures where the package can't be used to install the package.

  > Note: Be careful to use this **exclude_arch** field. If the package has different **exclude_arch** value in the different versions, the end user can install the package in the specific version without some arch values of **exclude_arch**.

- **Value**: (arch values are separated with a space. Please refer Appendix A: Platform and Arch Value Mapping Table to more information)
- **Default Value**: (Empty)
- **Example**: None
- **DSM Requirement**: 6.0
- **Example**:

```
exclude_arch="bromolow cedarview".
```

Field Name: checksum

- **Description**: Contains MD5 string to verify the package.tgz.
- **Value**: String
- **Default Value**: (Empty)

- **Example**: None
- **DSM Requirement**: 3.2-1922

Field Name: adminport

- **Description**: A package listens to a specific port to display its own UI. If the package is defined by a port, a link will be opened when the package is started.
  > Note: **adminprotocol**, **adminport**and **adminurl** keys are combined to **adminprotocol**://ip:**adminport**/**adminurl** link
- **Value**: 0~65536
- **Default Value**: 80
- **Example**:

```
adminport="9002"
```

- **DSM Requirement**: 2.0-0731

Field Name: adminurl

- **Description**: If a package is installed and has a webpage, a link will be opened when the package is started.
  > Note: **adminprotocol**, **adminport**and **adminurl** keys are combined to **adminprotocol**://ip:**adminport**/**adminurl** link
- **Value**: String
- **Default Value**: (Empty)
- **Example**:

```
adminurl="web"
```

- **DSM Requirement**: 2.3-1118

Field Name: adminprotocol

- **Description**: A package uses a specific protocol to display its own UI. If a package is installed and has a webpage, a protocol will be opened when the package is started.
  > Note: **adminprotocol**, **adminport**and **adminurl** keys are combined to **adminprotocol**://ip:**adminport**/**adminurl** link
- **Value**: http / https
  (Separated with a space)
- **Default Value**: http
- **Example**:

```
adminprotocol="http"
```

- **DSM Requirement**: 3.2-1922

Field Name: dsmuidir

- **Description**: DSM UI folder name in package.tgz. The UI folder of the package in **/var/packges/[packge name]/target/[dsmuidir]** will be automatically linked to the DSM UI folder in **/usr/syno/synoman/webman/3rdparty/[package name]** to show your package's shortcut in DSM after the package is started. To remove the link, after the package is stopped.

> Note:
>
> 1. This key cannot contain : or /.
> 2. Please refer Integrate Your package into DSM for more information.

- **Value**: String
- **Default Value**: (Empty)
- **Example**: None
- **DSM Requirement**: 3.2-1922

Field Name: dsmappname

- **Description**: The value of each individual application will be equal to the unique property name in DSM's config file so as to be integrated into Synology DiskStation.
  > Note: Please refer Config in Integrate Your package into DSM chapter for more information.
- **Value**: (Separated with a space)
- **Default Value**: (Empty)
- **Example**:

```
dsmappname="SYNO.SDS.PhotoStation SYNO.SDS.PersonalPhotoStation"
```

- **DSM Requirement**: 3.2-1922

Field Name: checkport

- **Description**: Check if there is any conflict between the **adminport** and the ports which are reserved or are listening on DSM except web-service ports (e.g. 80, 443) and DSM ports (e.g. 5000, 5001).
- **Value**: "yes"/"no"
- **Default Value**: "yes"
- **Example**: None
- **DSM Requirement**: 3.2-1922

Field Name: startable

- **Description**: When no program in the package provides the end-user with the options to enable or disable its function. This key is set to "no" and the end-user cannot start or stop the package in Package Center.
  > Note: Deprecated after 6.1-14907, use *ctl_stop* instead.
  >
  > If "startable" is set to "no", **start-stop-status** script which runs in bootup or shotdown is still required.
- **Value**: "yes"/"no"
- **Default Value**: "yes"
- **Example**: None
- **DSM Requirement**: 3.2-1922

Field Name: ctl_stop

- **Description**: When no program in the package provides the end-user with the options to enable or disable its function. This key is set to "no" and the end-user cannot start or stop the package in Package Center.
  > Note: If "ctl_stop" is set to "no", **start-stop-status** script which runs in bootup or shotdown is still required.
- **Value**: "yes"/"no"
- **Default Value**: "yes"
- **Example**: None

- **DSM Requirement**: 6.1-14907

Field Name: ctl_uninstall

- **Description**: If this key is set to "no", the end-user cannot uninstall the package in Package Center.
- **Value**: "yes"/"no"
- **Default Value**: "yes"
- **Example**: None
- **DSM Requirement**: 6.1-14907

Field Name: precheckstartstop

- **Description**: If set to "yes", let start-stop-status with prestart or prestop argument run before start or stop the package. Please refer to start-stop-status in scripts for more information.
- **Value**: "yes"/"no"
- **Default Value**: "yes"
- **Example**: None
- **DSM Requirement**: 6.0

Field Name: helpurl

- **Description**: If a package is installed and has a "help" webpage, Package Center will display a hyperlink to the user.
- **Value**: String
- **Default Value**: (Empty)
- **Example**:

```
helpurl="https://www.synology.com/en-global/knowledgebase"
```

- **DSM Requirement**: 3.2-1922

Field Name: beta

- **Description**: If this package is considered the beta version, the beta information will be shown in Package Center.
- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**: None
- **DSM Requirement**: 6.0

Field Name: report_url

- **Description**: If a package is a beta version and has a "report" webpage, Package Center will display a hyperlink. If this package is considered the beta version, the beta information will be also be shown in Package Center.
- **Value**: String
- **Default Value**: (Empty)
- **Example**: None
- **DSM Requirement**: 3.2-1922

Field Name: install_reboot

- **Description**: Reboot DiskStation after installing or upgrading the package.
- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**: None
- **DSM Requirement**: 3.2-1922

Field Name: install_dep_packages

- **Description**: Before a package is installed or upgraded, these packages must be installed first. In addition, the order of starting or stopping packages is also dependent on it. The format consists of a package name. If more than one dependent packages are required, the package name of the package(s) will be separated with a colon, e.g. **install_dep_packages="packageA"**. If a specific version range is required, package name will be followed by one of the special characters =, <, >, >=, <= and package version which is composed by number and periods, e.g. **install_dep_packages="packageA>2.2.2:packageB"**.
  > Note: >= and <= operator only supported in DSM 4.2 or newer. Don't use <= and >= if a package can be installed in DSM 4.1 or older because it cannot be compared correctly. Instead, the package version should be set lower or higher.
- **Value**: Package names
  > Note: Each package name is separated with a colon.
- **Default Value**: (Empty)
- **Example**:

```
install_dep_packages="packageA"
or
install_dep_packages="packageA>2.2.2:packageB"
```

- **DSM Requirement**: 3.2-1922

Field Name: install_conflict_packages

- **Description**: Before your package is installed or upgraded, these conflict packages cannot be installed. The format consists of a package name, e.g. **install_conflict_packages="packageA"**. If more than one conflict packages are required with the format, the name of the package(s) will be separated with a colon, e.g. **install_conflict_packages="packageA:packageB"**. If a specific version range is required, package name will be followed by one of the special characters =, <, >, >=, <= and package version which is composed by number and periods, e.g. **install_conflict_packages="packageA>2.2.2:packageB"**.
  > Note: >= and <= operator only supported in DSM 4.2 or newer. Do not use <= and >= if a package can be installed in DSM 4.1 because it can't be compared correctly. Instead, the package version should be set lower or higher.
- **Value**: Package names
  > Note: Each package name is separated with a colon.
- **Default Value**: (Empty)
- **Example**:

```
install_conflict_packages="packageA:packageB"
or
install_conflict_packages="packageA>2.2.2:packageB"
```

- **DSM Requirement**: 4.1-2851

Field Name: install_break_packages

- **Description**: After your package is installed or upgraded, these to-be-broken packages will be stopped and remain broken during the existence of your package. The format consists of a package name, e.g. **install_break_packages="packageA"**. If more than one to-be-broken packages are required with the format, the name of the package(s) will be separated with a colon, e.g.

install_break_packages="packageA:packageB". If a specific version range is required, package name will be followed by one of the special characters =, <, >, >=, <= and package version which is composed by number and periods, e.g. **install_break_packages="packageA>2.2.2:packageB"**.

- **Value**: Package names
  > Note: Each package name is separated with a colon.
- **Default Value**: (Empty)
- **Example**:

```
install_break_packages="packageA:packageB"
or
install_break_packages="packageA>2.2.2:packageB"
```

- **DSM Requirement**: 6.1-15117

Field Name: install_replace_packages

- **Description**: After your package is installed or upgraded, these to-be-replaced packages will be removed. The format consists of a package name, e.g. **install_replace_packages="packageA"**. If more than one to-be-replaced packages are required with the format, the name of the package(s) will be separated with a colon, e.g. **install_replace_packages="packageA:packageB"**. If a specific version range is required, package name will be followed by one of the special characters =, <, >, >=, <= and package version which is composed by number and periods, e.g. **install_replace_packages="packageA>2.2.2:packageB"**.
- **Value**: Package names
  > Note: Each package name is separated with a colon.
- **Default Value**: (Empty)
- **Example**:

```
install_replace_packages="packageA:packageB"
or
install_replace_packages="packageA>2.2.2:packageB"
```

- **DSM Requirement**: 6.1-15117

Field Name: instuninst_restart_services

- **Description**: Reload services after installing, upgrading and uninstalling the package.
  > Note:
  >
  > 1. If the service is not enabled or started by the end-user, services won't be reloaded
  > 2. If the **install_reboot** is set to “yes”, this value is ignored in the installation process.
- **Value**:
  DSM 4.3 or older: apache-sys, apache-web, mdns, samba, db, applenetwork, cron, nfs, firewall
  DSM 5.0 ~ DSM 5.2: apache-sys, apache-web, mdns, samba, applenetwork, cron, nfs, firewall
  DSM 6.0: nginx, mdns, samba, applenetwork, cron, nfs, firewall
  > Note: Each service is separated with a space.
- **Default Value**: (Empty)
- **Example**:

```
instuninst_restart_services="apache-sys apache-web"
```

- **DSM Requirement**: 3.2-1922

Field Name: startstop_restart_services

- **Description**: Reload services after starting and stopping the package.
  > Note:
  >
  > 1. If the service is not enabled or started by the end-user, services won't be reloaded.
  > 2. If startable is set to “no”, the value is ignored.
- **Value**:
  DSM 4.3 or older: apache-sys, apache-web, mdns, samba, db, applenetwork, cron, nfs, firewall
  DSM 5.0 ~ DSM 5.2: apache-sys, apache-web, mdns, samba, applenetwork, cron, nfs, firewall
  DSM 6.0: nginx, mdns, samba, applenetwork, cron, nfs, firewall
  > Note: Each service is separated with a space.
- **Default Value**: (Empty)
- **Example**:

```
startstop_restart_services="apache-sys apache-web"
```

- **DSM Requirement**: 3.2-1922

Field Name: install_dep_services

- **Description**: Before the package is installed or upgraded, these services must be started or enabled by the end-user.
- **Value**:
  DSM 4.2 or older: apache-web, mysql, php_disable_safe_exec_dir
  DSM 4.3: apache-web, mysql, php_disable_safe_exec_dir, ssh
  DSM 5.0 ~ DSM 5.2: apache-web, php_disable_safe_exec_dir, ssh, pgsql
  DSM 6.0: ssh, pgsql
  > Note: Each service is separated with a space.
- **Default Value**: (Empty)
- **Example**:

```
install_dep_services="apache-web ssh"
```

- **DSM Requirement**: 3.2-1922

Field Name: start_dep_services

- **Description**: Before the package is started, these services must be started or enabled by the end-user. If startable is set to “no”, this value is ignored.
- **Value**:
  DSM 4.2 or older: apache-web, mysql, php_disable_safe_exec_dir
  DSM 4.3: apache-web, mysql, php_disable_safe_exec_dir, ssh
  DSM 5.0 ~ DSM 5.2: apache-web, php_disable_safe_exec_dir, ssh, pgsql
  DSM 6.0: ssh, pgsql
  > Note: Each service is separated with a space.
- **Default Value**: (Empty)
- **Example**:

```
install_dep_services="apache-web ssh"
```

- **DSM Requirement**: 3.2-1922

Field Name: extractsize

- **Description**: This value indicates the minimal space to install a package. It will be used to prompt the user if there is enough free space to install it.
  > Note:
  >
  > 1. In DSM 5.2 or order, the size based on byte unit.
  > 2. In DSM 6.0 or newer, the size based on kilobyte unit.
- **Value**: Size unit
- **Default Value**: The byte size of SPK file of package
- **Example**:

```
extractsize="253796"
```

- **DSM Requirement**: 4.0-2166

Field Name: support_conf_folder

- **Description**: In DSM 5.2 or order, if you want to use some special configuration files within a "**conf**" folder, this value must be set to "**yes**". More details are given in the "**conf**" section. Howerver, in DSM 6.0 or newer, you don't need to define it anymore.
  > Note: Deprecated in DSM 6.0
- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**:

```
support_conf_folder="yes"
```

- **DSM Requirement**: 4.2-3160 ~ 5.2

Field Name: install_type

- **Description**: If set to “system”, your package will be installed in the root file system, **/usr/local/packages/@appstore/**, even if there is no volume.
  > Note: Be careful when setting this, as it may result in the DiskStation crashing if your package runs out of the space in the root file system.
- **Value**: "system"
- **Default Value**: (Empty)
- **Example**:

```
install_type="system"
```

- **DSM Requirement**: 5.0-4458

Field Name: silent_install

- **Description**: If set to “yes”, your package is allowed to be installed without the package wizard in the background. This allows CMS (Central Management System) to distribute package installation to other NAS connected.
- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**:

```
silent_install="yes"
```

- **DSM Requirement**: 5.0-4458

Field Name: silent_upgrade

- **Description**: If set to “yes”, your package is allowed to be upgraded without the package wizard in the background. End user cannot modify any information for upgrading. This allows not only your package to be upgraded automatically but also for CMS (Central Management System) to distribute package upgrades to other NAS connected.
- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**:

```
silent_upgrade="yes"
```

- **DSM Requirement**: 5.0-4458

Field Name: silent_uninstall

- **Description**: If set to “yes”, your package is allowed to be uninstalled without the package wizard in the background. This allows CMS (Central Management System) to distribute package uninstallation to other NAS connected.
- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**:

```
silent_uninstall="yes"
```

- **DSM Requirement**: 5.0-4458

Field Name: auto_upgrade_from

- **Description**: It is set to a version of your package. If your package is set to **silent_upgrade="yes"** and the value is set, Package Center only upgrades your package automatically from the installed package with the version or the newer version. However, if the end user install a older version than it, Package Center won't upgrade it automatically and the user must upgrade it by themself.
- **Value**: (a package version)
- **Default Value**: (Empty string)
- **Example**:

```
auto_upgrade_from="2.0"
```

- **DSM Requirement**: 5.2-5565

Field Name: offline_install

- **Description**: If set to "yes", after the package is published in synology server, it won't be shown in the package list of Package Center from Synology server. However, the user can install the package manually.
- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**:

```
offline_install="yes"
```

- **DSM Requirement**: DSM 6.0

Field Name: thirdparty

- **Description**: If set to “yes”, your package is a third-party package and isn't developed by Synology. In Package Center, third-pary pacakges will be shown in another part.

  > Note: It's not used in DSM 5.0 or newer.

- **Value**: "yes"/"no"
- **Default Value**: "no"
- **Example**:

```
thirdparty="yes"
```

- **DSM Requirement**: 4.0~4.3

Field Name: os_max_ver

- **Description:** Maximum version of DSM that is capable to run the package.
- **Value:** X.Y-Z DSM major number, DSM minor number, DSM build number
- **Default Value:** None
- **Example:**

```
os_max_ver="6.1-14715"
```

- **DSM Requirement:** 6.1-14715

package.tgz

The **package.tgz** is a compressed file containing all the files you would need when building up your applications. For example,

- executable files
- libraries
- UI files
- configuration files

You can use **pkg_make_package** function to create the **package.tgz**. For more details about how to create the package.tgz, please refer to
Pack Stage: SynoBuildConf/install.

After you install your package, Package Center will extract package.tgz to **@appstore** folder (**/volume?/@appstore/** with the assigned volume or **/usr/local/packages/@appstore/**). In addition, Package Center will also create a soft link in **/var/packages/[package identity]/target** and point to the assigned folder.

> Note:
>
> 1. In DSM 5.2 or older, package.tgz must be tgz format.
> 2. In DSM 6.0 or newer, package.tgz can be tgz or xz format, but the file name must be package.tgz.

scripts

This folder contains shell scripts which will be executed during the installation, un-installation, upgrading, starting, and stopping of packages. Package Center will put the scripts to **/var/packages/[package identify]/scripts/** after the package is installed. There are seven basic script files stored in the **scripts** folder.

1. **preinst**: This script is run before the package files are transferred to **@appstore**. You can check if the installation requirements meet the DSM or package version, or if some services are enabled in this script.
   For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Hello!!" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.

2. **postinst**: This script is run after the package files are transferred to **@appstore**. You can change the file permission and ownership in this script.
   For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Hello!!" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.

3. **preuninst**: This script is run before the package is removed.
   For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Hello!!" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.

4. **postuninst**: This script is run after the package is removed from the system.
   For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Hello!!" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.

5. **preupgrade**: Package Center will call this script before uninstalling the old version of your package.
   For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user(e.g. `echo "Hello!!" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.

6. **postupgrade**: Package Center will call this script after installing the new version of your package.
   For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user(e.g. `echo "Hello!!" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.

7. **start-stop-status**: This script is used to start and stop a package, detect running status, and generate the log file. Parameters used by the script are listed in below:

   i. **start**: When the user clicks "**Run**" to run the package or the NAS is turned on, the Package Center will call this script with the "**start**" parameter. A returned value will then be acquired.
      For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Start failed" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.
   ii. **stop**: When the user clicks "**Stop**" to stop running the package or the NAS is turned off, the Package Center will call this script with the "**stop**" parameter. A returned value will then be acquired.
      For non-zero returned values, you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Stop failed" > $SYNOPKG_TEMP_LOGFILE`). You can also write messages in the **SYNOPKG_TEMP_LOGFILE** file for zero returned values which represent that the process was successful.
   iii. **status**: When Package Center is opened to check package status, it will send a request to ask the status of the package using this parameter. The following exit status codes should be returned:

```
0: package is running.
1: program of package is dead and /var/run pid file exists.
2: program of package is dead and /var/lock lock file exists
3: package is not running
4: package status is unknown
150: package is broken and should be reinstalled. Please note, broken status (150) is only supported by DSM 4.2
```

iv. **log**: When a log page is opened in Package Center, Package Center will send a request to ask the log of the package using this parameter. When the log file name is sent to **STDOUT**, the content of the log file will be displayed.

v. **prestart**: If **precheckstartstop** in INFO is set to "yes", it is run for checking if it's allowed to the end user to start your package in some situations or not. For zero returned value, the end user can start your package. For non-zero returned values, the end user isn't allowed to start your package and you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Start failed" > $SYNOPKG_TEMP_LOGFILE`)

> Note:
>
> i. It only works in DSM 6.0 or newer.
> ii. It won't run after starting a package at booting up.

vi. **prestop**: If **precheckstartstop** in INFO is set to "yes", it is run for checking if it's allowed to the end user to stop your package in some situations or not. For zero returned value, the end user can stop your package. For non-zero returned values, the end user isn't allowed to stop your package and you can compile error messages in the **SYNOPKG_TEMP_LOGFILE** file to prompt the user (e.g. `echo "Stop failed" > $SYNOPKG_TEMP_LOGFILE`)

> Note:
>
> i. It only works in DSM 6.0 or newer.
> ii. It won't run before stopping a package at shutting down.

To install a package:

- preinst
- postinst
- start-stop-status with prestart argument if end user chooses to start it immediately
- start-stop-status with start argument if end user chooses to start it immediately

To upgrade a package:

- start-stop-status with prestop argument if it has been started
- start-stop-status with stop argument if it has been started
- preupgrade
- preuninst
- postuninst
- preinst
- postinst
- postupgrade
- start-stop-status with prestart argument if it was started before being upgraded
- start-stop-status with start argument if it was started before being upgraded

To uninstall a package:

- start-stop-status with prestop argument if it has been started
- start-stop-status with stop argument if it has been started
- preuninst
- postuninst

To start a package:

- start-stop-status with prestart argument
- start-stop-status with start argument

To stop a package:

- start-stop-status with prestop argument
- start-stop-status with stop argument

> Note: start-stop-status with prestart or prestop argument is only supported in DSM 6.0 or newer.

Script Environment Variables

Several variables are exported by Package Center and can be used in the scripts. Descriptions of the variables are given as below:

- **SYNOPKG_PKGNAME**: Package identify which is defined in **INFO**.
- **SYNOPKG_PKGVER**: Package version which is defined in **INFO**.
- **SYNOPKG_PKGDEST**: Target directory where the package is stored.
- **SYNOPKG_PKGDEST_VOL**: Target volume where the package is stored.

  > Note: It's only available in DSM 4.2 or above.

- **SYNOPKG_PKGPORT**: **adminport** port which is defined in **INFO**. This port will be occupied by this package with its management interface.
- **SYNOPKG_PKGINST_TEMP_DIR**: The temporary directory where the package are extracted when installing or upgrading it.
- **SYNOPKG_TEMP_LOGFILE**: A temporary file path for a script to log information or error messages.
- **SYNOPKG_TEMP_UPGRADE_FOLDER**: The temporary directory when the package is upgrading. You can move the files from the previous version of the package to it in **preupgrade** script and move them back in **postupgrade**.

  > Note: It's only available in DSM 6.0 or above.

- **SYNOPKG_DSM_LANGUAGE**: End user's DSM language.
- **SYNOPKG_DSM_VERSION_MAJOR**: End user's major number of DSM version which is formatted as [DSM major number].[DSM minor number]-[DSM build number].
- **SYNOPKG_DSM_VERSION_MINOR**: End user's minor number of DSM version which is formatted as [DSM major number].[DSM minor number]-[DSM build number].
- **SYNOPKG_DSM_VERSION_BUILD**: End user's DSM build number of DSM version which is formatted as [DSM major number].[DSM minor number]-[DSM build number].
- **SYNOPKG_DSM_ARCH**: End user's DSM CPU architecture. Please refer Appendix A: Platform and Arch Value Mapping Table to more information
- **SYNOPKG_PKG_STATUS**: Package status presented by these values: INSTALL, UPGRADE, UNINSTALL, START, STOP or empty.
  1. INSTALL will be set as the status value in the **preinst** and **postinst** scripts while the package is installing. If the user chooses to "start after installation" at the last step of the installation wizard, the value will be set to INSTALL in the **start-stop-status** script when the package is started.
  2. UPGRADE will be set as the status value in the **preupgrade**, **preuninst**, **postunist**, **preinst**, **postinst** and **postupgrade** scripts sequentially while the package is upgrading. If the package has already started before upgrade, the value will be set to UPGRADE in the **start-stop-status** script when the package is started or stopped.
  3. UNINSTALL will be set as the status value in the **preuninst** and **postunist** scripts while the package is un-installing. If the package has already started before un-installation, the value will be set to UNINSTALL in the **start-stop-status** script when the package is stopped.
  4. If the user starts or stops a package in the Package Center, START or STOP will be set as the status value in the **start-stop-status** script.
  5. When the NAS is booting up or shutting down, its status value will be empty.

     > Note: **SYNOPKG_PKG_STATUS** is only available for the **start-stop-status** script in DSM 4.0 or above.

- **SYNOPKG_OLD_PKGVER**: Existing package version which is defined in INFO (only in **preupgrade** script).
- **SYNOPKG_TEMP_SPKFILE**: The location of package spk file is temporarily stored in DS when the package is installing/upgrading.

  > Note: It's only available in DSM 4.2 or above.

- **SYNOPKG_USERNAME**: The user name who installs, upgrades, uninstalls, starts or stops the package. If the value is empty, the action is triggered by DSM, not by the end user.

  > Note: It's only available in DSM 5.2 or above.

- **SYNOPKG_PKG_PROGRESS_PATH**: A temporary file path for a script to showing the progress in installing and upgrading a package.

> Note:
>
> 1. The progress value is between 0 and 1.
> 2. It's only available in DSM 5.2 or above.
> 3. Example:
>
> ```
> flock -x "$SYNOPKG_PKG_PROGRESS_PATH" -c echo 0.80 > "$SYNOPKG_PKG_PROGRESS_PATH"
> ```

Once the end user enters or selects some values of the UI components which are configured in **install_uifile(.sh)/upgrade_uifile(.sh)/uninstall_uifile(.sh)** (Please refer to WIZARD_UIFILES section for more information), the names and values of the components will be set in the environment variables. Also note that the names of these components cannot be the same as those of the environment variables.

conf

The "**conf**" folder contains special configurations including some information which cannot be described in the **INFO** file with key/pair format. Package Center controls the flow of installation, upgrading, un-installation, starting, and stopping processes according to these configurations.

In DSM 4.2, there are two configurations, **PKG_DEPS** and **PKG_CONX**, which are stored in this folder. They are used to define dependency or conflict between the packages. The dependency or conflict will be checked according to the end user's DSM version. For example, Perl was built on DSM 4.1, but it does not exist on DSM 4.2. Therefore, if your package depends on Perl, the Perl package must be installed on DSM 4.2 before your package can be installed. You can set the **PKG_DEPS** configuration to indicate that the dependency rule only works on DSM 4.2 or later.

The dependency or conflict is similar to **install_dep_packages** and **install_conflict_packages** keys in **INFO** file, but they do not define the restriction according to specific DSM versions.

PKG_DEPS and **PKG_CONX** always have higher priority compared with the keys in the **INFO** file. That is, if you define dependency in the **PKG_DEPS** file, then the **install_dep_packages** key in the **INFO** file will be ignored in DSM 4.2 or later. If you define the conflict in the **PKG_CONX** file, then the **install_conflict_packages** key in the **INFO** file will be ignored in DSM 4.2 or later.

The **conf** folder contains the following files:

File/Folder Name	Description	File/Folder Type	DSM Requirement
PKG_DEPS	Define dependency between packages with restrictions of DSM version. Before your package is installed or upgraded, these packages must be installed first. Package Center controls the order of start or stop packages according to the dependency.	File	4.2-3160
PKG_CONX	Define conflicts between packages with restrictions of DSM version. Before your package is installed or upgraded, these conflicting packages cannot be installed.	File	4.2-3160

> Note: All words are case sensitive.

Each configuration file is defined in standard **.ini** file format with key/value pairs and sessions, for example:

```
[session]
```

A session describes a unique name of dependent/conflicting package. Each session contains information about the requirements of package versions and the restriction of DSM versions.

- Keys configured in **PKG_DEPS** file each dependent package (session) contains:

Key	Description	Value
pkg_min_ver	Minimum version of dependent package. You must install this dependent package with this version or newer before installing your package.	Package version
pkg_max_ver	Maximum version of dependent package. You must install this dependent package with the version or older before installing your package.	Package version
dsm_min_ver	Minimum required DSM version. If you have this version or newer of DSM, this dependency will be considered, but it will be ignored in an older DSM.	X.Y-Z DSM major number, DSM minor number, DSM build number
dsm_max_ver	Maximum required DSM version. If you have this version or older of DSM, this dependency will be considered, but it will be ignored in a newer DSM.	X.Y-Z DSM major number, DSM minor number, DSM build number

Example:

```
; Your package depends on Package A in any version
[Package A]

; Your package depends on Package B version 2 or newer
pkg_min_ver=2

; Your package depends on Package C with version 2 or older
[Package C]
pkg_max_ver=2

; Your package depends on Package D with version 2 or older but it will be ignored when DSM version is older than 4.1-2(
[Package D]
dsm_min_ver=4.1-2668
pkg_min_ver=2

; Your package depends on Package E with version 2 or newer but it will be ignored when DSM version is newer than 4.1-2(
[Package E]
dsm_max_ver=4.1-2668
pkg_min_ver=2
```

- Keys configured in **PKG_CONX** file each conflicting package (session) contain:

Key	Description	Value
pkg_min_ver	Minimum version of conflicting package. If end user installs this conflicting package with the specified version or newer, he will not be able to install your package.	Package Version
pkg_max_ver	Maximum version of conflicting package. If end user installs this conflicting package with the specified version or older, he will not be able to install your package.	Package Version
dsm_min_ver	Minimum requried DSM version. If end user has the specified version or newer of DSM, this conflict will be considered, but it will be ignored in older versions of DSM.	X.Y-Z DSM major number, DSM minor number, DSM build number
dsm_max_ver	Maximum requried DSM version. If the end user has the specified version or older of DSM, this conflict will be considered, but it will be ignored in newer DSM.	X.Y-Z DSM major number, DSM minor number, DSM build number

Example:

```
; Your package conflicts with Package A in any version
[Package A]

; Your package conflicts with Package B version 2 or newer
[Package B]
pkg_min_ver=2

; Your package conflicts with Package C version 2 or older
[Package C]
pkg_max_ver=2

; Your package conflicts with Package D version 2 or older, but it will be ignored when DSM version is older than 4.1-2(
[Package D]
dsm_min_ver=4.1-2668
pkg_min_ver=2

; Your package conflict on Package E with version 2 or newer but it will be ignored when DSM version is newer than 4.1-2
[Package E]
dsm_max_ver=4.1-2668
pkg_min_ver=2
```

WIZARD_UIFILES

install_uifile, **upgrade_uifile**, and **uninstall_uifile** are files which describe UI components in JSON format. They are stored in the "**WIZARD_UIFILES**" folder. During the installation, upgrading, and un-installation processes, these UI components will appear in the wizard. Once these components are selected, their keys will be set in the script environment variables with true, false, or text values.

These files can be regarded as user settings or used to control the flow of script execution.

- **install_uifile**: Describes UI components for the installation process. During the process of the **preinst** and **postinst** scripts, these component keys and values can be found in the environment variables.
- **upgrade_uifile**: Describes UI components for the upgrade process. During the process of the **preupgrade**, **postupgrade**, **preuninst**, **postuninst**, **preinst** and **postinst** scripts, these component keys and values can be found in the environment variables.
- **uninstall_uifile**: Describes UI components for the un-installation process. During the process of the **preuninst** and **postuninst** scripts, these component keys and values can be found in the environment variables.

If you would like to run a script to generate the wizard dynamically, you can add **install_uifile.sh**, **upgrade_uifile.sh** and **uninstall_uifile.sh** files, they are run before installing, upgrading, and uninstalling a package respectively to generate UI components in JSON format and write to **SYNOPKG_TEMP_LOGFILE**. Script environment variables in these scripts can be gotten in these scripts. Please refer to "Script Environment Variables" for more information.

If you would like to localize the descriptions of UI components, you can add a language abbreviation suffix to the file "**install_uifile_[DSM language]**," **"upgrade_uifile_[DSM language]**", "**uninstall_uifile_[DSM language]**", "**install_uifile_[DSM language]**.sh," **"upgrade_uifile_[DSM language]**.sh" or "**uninstall_uifile_[DSM language]**.sh" in this folder. For example, in order to perform installation in Traditional Chinese, **[DSM language]** should be replaced with "**cht**" as follows: "**install_uifile_cht**".

Example of the file in JSON format:

```
[{
    "step_title": "Step1",
    "items": [{
        "type": "singleselect",
        "desc": "a radio group",
        "subitems": [{
            "key": "radio1",
            "desc": "Radio button 1",
            "defaultVaule": false
        }, {
            "key": "radio2",
            "desc": "Radio button 2",
            "defaultVaule": true
        }]
    }]
}, {
    "step_title": "Step2",
    "items": [{
        "type": "multiselect",
        "desc": "a check group",
        "subitems": [{
            "key": "check1",
            "desc": "Check button 1"
        }, {
            "key": "check2",
            "desc": "Check button 2",
            "defaultVaule": true,
            "validator": {
                "fn": "{var v=arguments[0]; if (!v) return 'Check this';return true;}"
             }
        }]
    }, {
        "type": "textfield",
        "desc": "textfield",
        "subitems": [{
            "key": "textfield1",
```

```
            "desc": "textfield 1",
            "defaultVaule": "default",
            "validator": {
                "allowBlank": false,
                "minLength": 2,
                "maxLength": 10
            }
        },{
            "key": "textfield2",
            "desc": "textfield 2",
            "emptyText": "abc1@cde.com",
            "validator": {
                "vtype": "email",
                "regex": {
                    "expr": "/[0-9]/i",
                    "errorText": "Error"
                }
            }
        }]
    }]
}, {
    "step_title" : "Step 3",
    "invalid_next_disabled": true,
    "activeate": "{console.log('activeate', arguments);}",
    "deactivate": "{console.log('deactivate', arguments);}",
    "items" : [{
        "type" : "singleselect",
        "desc" : "Check it",
        "subitems": [{
                "key": "id1",
                "desc": "Not choose it",
                "defaultValue": true
            },
            {
                "key": "id2",
                "desc": "Choose it",
                "defaultValue": false,
                "validator": {
                    "fn": "{return arguments[0];}"
                }
            }]
        }]
}]
```

Example of using a script to generate a file in JSON format:

```
#!/bin/sh

/bin/cat > /tmp/wizard.php <<'EOF'
<?php
$ini_array = parse_ini_file("/etc.defaults/synoinfo.conf");
$unique=$ini_array["unique"];
echo  <<<EOF
[{
    "step_title": "Step 1",
    "items": [{
        "type": "textfield",
        "desc": "model name",
        "subitems": [{
            "key": "pkgwizard_db_name",
            "desc": "name",
            "defaultValue":"$unique"
        }]
    }, {
        "type": "combobox",
        "desc": "Please select a volume",
        "subitems": [{
            "key": "volume",
            "desc": "volume name",
            "displayField": "display_name",
            "valueField": "volume_path",
            "editable": false,
            "mode": "remote",
            "api_store": {
                "api": "SYNO.Core.Storage.Volume",
                "method": "list",
                "version": 1,
                "baseParams": {
                    "limit": -1,
                    "offset": 0,
                    "location": "internal"
                },
                "root": "volumes",
                "idProperty": "volume_path",
                "fields": ["display_name", "volume_path"]
            },
            "validator": {
                "fn": "{console.log(arguments);return true;}"
            }
        }]
    }]
}];
EOF;
?>
EOF

/usr/bin/php -n /tmp/wizard.php > $SYNOPKG_TEMP_LOGFILE
rm /tmp/wizard.php
exit 0
```

- Here are the properties for each step in the wizard in JSON format:

Property	Description	DSM Requirement
step_title	Optional. Describes the title of the current step performed in the wizard.	3.2-1922
items	Describes an array containing the components of "**singleselect**", "**multiselect**", "**textfield**", "**password**", or "**combobox**" type.	3.2-1922
type	Must be "**singleselect**", "**multiselect**", "**textfield**" "**password**" or "**combobox**". "**singleselect**" type represents the components in the sub-items which are all radio buttons. You can select only one radio box with a unique key. "**multiselect**" type represents the components in the sub-items which are all checkboxes. You can check more than one checkbox. "**textfield**" type represents the components in the sub-items which are all text fields. You can type text. "**password**" type represents the components in the sub-items which are all password fields. You can type passwords. "**combobox**" type represents the components in the sub-items which are all combobox fields. The user can choose a item in the combobox field. Note: "**combobox**" type is only available in DSM 5.2 or newer.	3.2-1922
desc	Optional. Describe a component in the label text.	3.2-1922
subitems	Describe an array containing radio buttons, checkboxes, text fields, or password components.	3.2-1922
activeate	JSON-style string to describe a function which is run after the step of the wizard has been visually activated.	5.2
deactivate	JSON-style string to describe a function which is run after the step of the wizard has been visually deactivated.	5.2
invalid_next_disabled	If set to true, the next button in the step of the wizard will be disabled by default. It will be enabled if all items are validated scucuessfully by **validator** in this step.	5.2

- Here are the properties for components in **subitems** in JSON format:

Property	Description	DSM Requirement
key	A unique component key value represents a UI component. If you select a component, this key will be set in the script environment variables of **preinst**, **postinst**, **preupgrade**, **postupgrade**, **preuninst**, **postuninst**, **start-stop-status** (the string value of the selected checkbox or radio button is always "**true**".).	3.2-1922
defaultVaule	Optional. true/false value to initialize "**singleselect**" or "**multiselect**" component, or a string value to initialize "**textfield**" or "**password**" component.	4.2-3160
emptyText	Optional. The prompt text to place into an empty "**textfield**" or "**password**" component to prompt the user how to fill in if **defaultVaule** is not set.	4.2-3160
validator	JSON-style object to describe validation functions. If the validation fails with the user's value, the user cannot go to the next step of the wizard. More detailed properties of **validator** are given in the **validator** table.	4.2-3160
disabled	true to disable the field (defaults to false).	6.0
height	The height of this component in pixels.	6.0
hidden	true to hide this component.	6.0
invalidText	The error text to use when marking a field invalid and no message is provided.	6.0
preventMark	true to disable marking the field invalid. Defaults to false.	6.0
width	The width of this component in pixels.	6.0

- Here are the properties of **validator**:

Property	Description	Value
allowBlank	Specify false to validate that the value's length of "**textfield**" or "**password**" component is > 0	true/false
minLength	Minimum length of "**textfield**" or "**password**" component	Number
maxLength	Maximum length of "**textfield**" or "**password**" component	Number
vtype	Specify pre-defined validation function, "**alpha**": validate alpha value "**alphanum**": validate alphanumeric value "**email**": validate email address "**url**": validate URL	"alpha", "alphanum", "email", "url"
regex	Describe validation function in regular expression and invalid message. Properties contain: "**expr**": Javascript Regular Expression "**errorText**": invalid string	JSON-style object
fn	Describe the Javascript function which is encoded by JSON-style string with curly brackets. In this function, you can use arguments[0] to get the value of the component. In addition, this function will return true if the value is valid or as an invalid string if the value is invalid.	String

- Here are the other properties for **textfield**, **password** or **combobox** component in **subitems** in JSON format:

Property	Description	DSM Requirement
blankText	The error text to display if the allowBlank validation fails	6.0
grow	true if this field should automatically grow and shrink to its content	6.0
growMax	The maximum height to allow when **grow** is true	6.0
growMin	The minimum height to allow when **grow** is true	6.0
htmlEncode	false to skip HTML-encoding the text when rendering it (defaults to false).	6.0
maxLengthText	Error text to display if the maximum length validation using **maxLength** fails.	6.0
minLengthText	Error text to display if the minimum length validation using **minLength** fails.	6.0

- Here are the properties for **combobox** component in **subitems** in JSON format:

Property	Description	DSM Requirement

api_store	JSON-style object to describe to send a WebAPI request and store the response in the data strusture for combobox use. More detailed properties of **api_store** are given in the **store** table.
	Example:
	`{`
	`"api": "SYNO.Core.XXX",`
	`"method": "list",`
	`"version": 1,`
	`"baseParams": {`
	`"offset": 0,`
	`"limit":-1,`
	`},`
	`"root": "items",`
	`"idProperty":"name",`
	`"fields": ["name"]`
	`}`
		6.0

autoSelect	true to select the first result gathered by the data store (defaults to true). A false value would require a manual selection from the dropdown list to set the components value.	6.0
displayField	The underlying data field name to bind to this combobox.	6.0
editable	false to prevent the user from typing text directly into the field, the field will only respond to a click on the trigger to set the value. (defaults to true).	6.0
forceSelection	true to restrict the selected value to one of the values in the list, false to allow the user to set arbitrary text into the field (defaults to false).	6.0

handleHeight	The height in pixels of the dropdown list resize handle if **resizable** is true.	6.0
listAlign	A valid anchor position value.	6.0
listEmptyText	The empty text to display in the data view if no items are found.	6.0
listWidth	The width of the dropdown list.	6.0
maxHeight	The maximum height in pixels of the dropdown list before scrollbars are shown.	6.0
minChars	The minimum number of characters the user must type before autocomplete and **typeAhead** activate	6.0
minHeight	The minimum height in pixels of the dropdown list when the list is constrained by its distance to the viewport edges.	6.0
minListWidth	The minimum width of the dropdown list in pixels.	6.0
mode	Set to 'loacl' to load local **store** to load local JSON-array data. More detailed properties of local **store** are given in the **store** table. Example: `{ "mode": "local", "valueField": "myId", "displayField": "displayText", "store": { "xtype": "arraystore", "fields": ["myId", "displayText"], "data": [[1, "item1"], [2, "item2"]] } }`	6.0
pageSize	If greater than 0, a paging toolbar is displayed in the footer of the dropdown list and the filter queries will execute with page start and limit parameters. Only applies when using **api_store** (defaults to 0).	6.0
queryDelay	The length of time in milliseconds to delay between the start of typing and sending the query to filter the dropdown list.	6.0
resizable	true to add a resize handle to the bottom of the dropdown list (Defaults to false).	6.0
selectOnFocus	true to select any existing text in the field immediately on focus. Only applies when **editable** is true (defaults to false).	6.0
store	A data structure to store data in combobox (defaults to undefined). It can't be used with **api_store** at the same time. Acceptable values for this property are: 1-dimensional array : e.g., `["Foo","Bar"]` 2-dimensional array : For a 2-dimensional array, the value in index 0 of each item will be assumed to be the **valueField**, while the value at index 1 is assumed to be the **displayField**, e.g., `[["f","Foo"],["b","Bar"]]` .	6.0
title	If supplied, a header element is created containing this text and added into the top of the dropdown list	6.0
typeAhead	true to populate and autoselect the remainder of the text being typed after a configurable delay (**typeAheadDelay**).	6.0
typeAheadDelay	The length of time in milliseconds to wait until the typeahead text is displayed.	6.0
valueField	The underlying data value name to bind to this combobox.	6.0

- Here are the properties for **api_store** or **store** data structure in JSON format:

Property	Description	DSM Requirement
baseParams	An object containing properties which are to be sent as parameters for every WebAPI request in **api_store**.	6.0
data	An inline data object readable by the reader in local **store** to load local JSON-array data.	6.0
displayField	The underlying data field name to bind to this combobox.	6.0
fields	defined fields for the data stored in this **store** or **api_store**.	6.0
idProperty	Identity of the property within data that contains a unique value.	6.0
root	The name of the property which contains the array of data. Defaults to undefined.	6.0
valueField	The underlying data value name to bind to this combobox.	6.0
xtype	Only support **arraystore** type for local **store** to load local JSON-array data.	6.0

> Note:
>
> 1. All words are case sensitive.
> 2. In DSM 4.0 or above, if both the type and subitems properties are empty, text in the desc property will be displayed as one of the steps of wizard.
> 3. install_uifile.sh, upgrade_uifile.sh ,uninstall_uifile.sh and *.sh scripts to gernate the wizard dynamically are only supported in DSM 5.2 or newer.

Integrate Your Package into DSM

Manage Storage for Application Files

When a package is installing, **package.tgz** will be extracted to **/var/packages/[package identity]/target**, which is a symbolic link pointing to a folder in a data volume selected by the end user. **/var/packages/[package identity]/target** is also available at the **SYNOPKG_PKGDEST** environment variable, one of the seven files in the script folder. Please see the "scripts" for more information.

Despite the fact that the directory **/var/packages** or **/usr/local** is reserved for 3rd-party applications, the storage space of system volume is limited. If the size of files to be installed exceeds the capacity of the system volume, storage space will run out. Hence, it is recommended that you directly read or write application files in **/var/packages/[package identity]/target** or another space of the data volume. You can also make a symbolic link in **/usr/local** to point to **/var/packages/[package identity]/target** or another space when running the **postinst** script. The path can be accessed easier in a library or a daemon. Please note that you may need to specify the correct prefix when running a configuration script so that the application can find the correct path information upon execution.

Synology releases DSM updates on a regular basis. Given that application files might be affected during the update procedure, it is important that you install your application in the correct directory to prevent them from being deleted when DSM is being upgraded in the system partition.

When DSM is being upgraded, the directory **/var/packages/[package identity]** and **/usr/local** will be backed up and restored automatically. However, some library files or built-in software might be modified during the upgrade procedure. In other words, if your application depends on the files which are subject to change, the application may not work afterward. In this case, you should check the status of these files or re-link them in the **start-stop-status** script to repair them if necessary. Alternatively, you can install them directly to **/var/packages/[package identity]/target**.

Integrate Your Package into DSM Web GUI

With Synology DSM , integrating 3rd-party applications into your NAS is easy. When an application is integrated, its icon will appear in the Main Menu of DSM. Users can also use their own customized icons for these applications. To place an icon on DSM desktop, simply drag and drop it from the Main Menu to the DSM desktop.

Startup

To integrate an application into Synology DSM 3.0 or later, please follow the steps below:

1. Create a folder under the directory **/usr/syno/synoman/webman/3rdparty/**.
2. Create a text file named "**config**" in your folder.
3. Under the same directory, create a sub-folder named after your application, such as **/usr/syno/synoman/webman/3rdpaty/[package name]/**. Put all UI related components, such as images, CSS, and CGI in the directory.

In the **INFO** file, you can define the key **dsmuidir**, whose value is a DSM directory name in the **package.tgz** file. Package Center will automatically link/unlink it to **/usr/syno/synoman/webman/3rdparty/[package name]** based on the key when you start/stop the package. You should also define **dsmappname** in the **INFO** file to integrate the package with DSM applications. Please refer to **dsmuidir** and **dsmappname** key in Optional Fields section for more information to link the folder automatically.

Next we will discuss the details of UI configuration.

Config

The text file “**config**” is used to configure UI behavior. The content of **config** should be in **JSON** format.

For example:

```
{
    ".url": {
        "com.company.App1": {
            "type": "url",
            "allUsers": true,
            "title": "Test App1",
            "desc": "Description",
            "icon": "images/app_{0}.png",
            "url": "http://www.yahoo.com"
        },
        "com.company.App2": {
            "type": "legacy",
            "allUsers": true,
            "title": "Test App2",
            "desc": "Description 2",
            "icon": "images/app2_{0}.png",
            "url": "http://www.synology.com"
        }
    }
}
```

Details of **application.cfg** are stated in below.

Property	Description
com.company.App1 com.company.App2	In “**.url**”, each object should have a unique property name.
title (Required)	“**title**” represents the application name that will be displayed in the main menu.
desc	“**desc**” displays more details about this application upon mouse-over.

icon (Required)	“**icon**” indicates the icon for the application. It is a template string. The “{0}” can be replaced by “16”, “24”, “32”, “48”, “64”, “72”, “256” depending on the resolution of the icon.
	The icon must be saved under **/usr/syno/synoman/webman/3rdparty/xxx/** where xxx is the directory name of your package.
	For example, if you create a directory named "images" and put the icon image file “**icon.png**” in it, the full path for the icon would be:
	`/usr/syno/synoman/webman/3rdparty/xxx/images/icon_16.png`
	`/usr/syno/synoman/webman/3rdparty/xxx/images/icon_24.png`
	`/usr/syno/synoman/webman/3rdparty/xxx/images/icon_32.png`
	`/usr/syno/synoman/webman/3rdparty/xxx/images/icon_48.png`
	`/usr/syno/synoman/webman/3rdparty/xxx/images/icon_64.png`
	`/usr/syno/synoman/webman/3rdparty/xxx/images/icon_72.png`
	`/usr/syno/synoman/webman/3rdparty/xxx/images/icon_256.png`
	The icon value should also be set as "**images/icon_{0}.png**"

type (Required)	When you click the menu item, the address you use to connect to the DSM management UI will be shown in the right frame of the management UI. However, you can customize the address as you wish. The “type” value can be "**url**" or "**legacy**". "**url**" means when you click the application icon, the URL will be opened in a pop-up window, while "**legacy**" implies that the URL will be opened in an iframe window application. You can follow the descriptions below to set up your customized URL.
url (Required)	The following is an example of value setting for your URL of the application: “url”: http://www.synology.com/ “url”: “3rdparty/xxx/index.html”
allUsers	This key determines whether or not the menu items can be seen by users when they log in with an admin account. If you would like to have all users see the menu items, please set the key value as below: `"allUsers": true` The default setting is that only the **admin** can find the application.

Integrate Help Document into DSM Help

To integrate a help document of your application into DSM Help, please do the following steps:

- Classify the help documents according to language, and put them in the help folder of your application.
- To have consistent style, and our customized scroll bar, you should add the following html tag:

```
<!DOCTYPE html>
<head>
<meta charset="UTF-8" />
<meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">

<link href="../../../../help/help.css" rel="stylesheet" type="text/css">
<link href="../../../../help/scrollbar/flexcroll.css" rel="stylesheet" type="text/css">
<script type="text/javascript" src="../../../../help/scrollbar/flexcroll.js"></script>
<script type="text/javascript" src="../../../../help/scrollbar/initFlexcroll.js"></script>

</head>
```

> Note: The js files are required because the native browser scroll bar has been disabled.

- You will need to add a text file "helptoc.conf" into your application. This text file "helptoc.conf" is to configure the structure of your help document. The content of helptoc.conf should be in JSON format. For example:

```
{
   "app": "SYNO.App.TestAppInstance",
   "title": "app_tree:index_title"
   "helpset": "help",
   "stringset": "texts",
   "content": "testapp_index.html",
   "toc": [{
   "title": "app_tree:node_1"
   "content": "testapp_node1.html",
   "nodes": [{
       "title": "app_tree:node_1_child"
       "content": "testapp_node1_child.html"
}]
}, {
   "title": "app_tree:node_2"
   "content": "testapp_node2.html"
}]
}
```

Details of **helptoc.conf** are stated in below.

Property	Description
stringset (Required)	"stringset" is the folder that stores your application strings.
app	"app" represents the application instance.
helpset	"helpset" displays more details about the application upon mouse-over.
title (Required)	"title" is the text that will be displayed in the DSM Help tree. It consists of two parts - section and key, and is separated by a colon.
content (Required)	"content" represents the url of the node.
toc	"toc" are the child nodes for your application. (use empty array if your application doesn't have one)
nodes	"nodes" are the child nodes of toc. (not necessary if there is no child nodes)

- Add the following content to the resource specification file. Please refer to Index DB for more detail.

```
"indexdb": {
    "app-index"  : {
        "conf-relpath": "ui/index.conf",
        "db-relpath": "indexdb/appindexdb"
    },
    "help-index": {
        "conf-relpath": "ui/helptoc.conf",
        "db-relpath": "indexdb/helpindexdb"
    }
}
```

Integrate with DSM Web Authentication

After integrating your application into Synology DSM, you may want to perform an authentication check to ensure only logged-in users can access the page.

To check whether a user has logged in, run the CGI command in below.

```
/usr/syno/synoman/webman/modules/authenticate.cgi
```

The "**authenticate.cgi**" will output the user name if the user has logged in. There will be no output if the user has not been authenticated.

Below is an example:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <strings.h>

/**
 * Check whether user is logged in.
 *
 * If user has logged in, put the username into "user".
 *
 * @param user    The buffer for get username
 * @param bufsize The buffer size of user
 *
 * @return 0: User not logged in or error
 *         1: User logged in. The user name is written to given "user"
 */
int IsUserLogin(char *user, int bufsize)
{
    FILE *fp = NULL;
    char buf[1024];
    int login = 0;

    bzero(user, bufsize);

    fp = popen("/usr/syno/synoman/webman/modules/authenticate.cgi", "r");
    if (!fp) {
        return 0;
    }
    bzero(buf, sizeof(buf));
    fread(buf, 1024, 1, fp);


    if (strlen(buf) > 0) {
        snprintf(user, bufsize, "%s", buf);
        login = 1;
    }
    pclose(fp);

    return login;
}

int main(int argc, char **argv)
{
    char user[256];

    printf("Content-type: text/html\r\n\r\n");
    if (IsUserLogin(user, sizeof(user)) == 1) {
        printf("User is authenticated. Name: %s\n", user);
    } else {
        printf("User is not authenticated.\n");
    }
    return 0;
}
```

DSM might require a random value called **SynoToken** to prevent a CSRF(cross-site request forgery) attack after 4.3. When CSRF protection is enabled in the control panel, you must append **SynoToken** to the query string or header of the HTTP request.

In the query string:

```
http://192.168.1.1:5000/webman/3rdparty/DownloadStation/webUI/downloadman.cgi?SynoToken=9WuK4Cf50Vw7Q
```

In the request header:

```
X-SYNO-TOKEN:9WuK4Cf50Vw7Q
```

The value of SynoToken can be obtained from **login.cgi** if the user is logged in.

Request:

```
http://192.168.1.1:5000/webman/login.cgi
```

Response:

```
{"SynoToken": "9WuK4Cf50Vw7Q", "result": "success", "success": true}
```

If your application is based on ExtJs of DSM, please include **dsmtoken.cgi** in your header section.

```
<header>
 <script src="/webman/dsmtoken.cgi" > </script>
</header>
```

Once the **dsmtoken.cgi** is included, **Ext.Ajax.Request**, **Ext.data.Connection**, **Ext.form.basicForm** and **Ext.urlAppend** will append **SynoToken** to the HTTP request automatically.

```
<script>
 Ext.Ajax.Request({ … }) // add SynoToken at event 'beforerequest'
 Ext.data.Connection({ … }) // add SynoToken at event 'beforerequet'
 new Ext.form.basicForm({ … }) // add SynoToken at event 'beforeaction'
// Ext.urlAppend will add SynoToken internally
 url = Ext.urlAppend('http://192.168.1.1', Ext.urlEncode({ … }));
</script>
```

DSM Backward Compatibility

Weak link is a property of Apple's development framework which ensures backward compatibility. GCC has a similar property called "weak symbol." We utilize this capability to provide a weak link framework in **libsynosdk** for backward compatibility as well. You can find available headers in **usr/syno/include/libsynosdk** under chroot environment. Each function prototype in **synosdk/*_p.h** is labeled with a macro telling you when this function is added into **libsynosdk**. Therefore, you can input a function in DSM 4.2 as follows:

```
/* DO NOT include *_p.h directly */
#include <synosdk/user.h>
#include <synosdk/service.h>

/* example, SYNOServiceHomePathCheck is available since DSM 4.2 */
if (SYNOServiceHomePathCheck) {
    SYNOServiceHomePathCheck(szPath, TRUE, TRUE, &pResult);
} else {
    /* implement alternative to SYNOServiceHomePathCheck here */
}
```

As a result, when your application runs in DSM 4.2 and later, function **SYNOServiceHomePathCheck** in **libsynosdk.so** is invoked. In DSM 4.2 and older, **else-part** will be executed as a replacement to **SYNOServiceHomePathCheck**.

Show Messages to Users

If you want to prompt users with a message before they install, upgrade, or un-install a package in Package Center, you can write these messages in the **desc** key in **install_uifile**, **upgrade_uifile**, or **uninstall_uifile**. Please refer to "WIZARD_UIFILES" for more information.

If you want to send a prompt message to users after they install, upgrade, un-install, start, or stop a package in Package Center, you can implement them into the **$SYNOPKG_TEMP_LOGFILE** variable in the related scripts. For example,

```
echo "Hello World!!" > $SYNOPKG_TEMP_LOGFILE
```

If you want to prompt users in the language specified in their DSM settings, you can refer to the **$SYNOPKG_DSM_LANGUAGE** variable for language abbreviation as shown in the scripts below:

```
case $SYNOPKG_DSM_LANGUAGE in
        chs)
            echo ""     > $SYNOPKG_TEMP_LOGFILE
        ;;
        cht)
            echo ""     > $SYNOPKG_TEMP_LOGFILE
        ;;
        csy)
            echo "Český" > $SYNOPKG_TEMP_LOGFILE
        ;;
        dan)
            echo "Dansk" > $SYNOPKG_TEMP_LOGFILE
        ;;
        enu)
            echo "English" > $SYNOPKG_TEMP_LOGFILE
        ;;
        fre)
            echo "Français" > $SYNOPKG_TEMP_LOGFILE
        ;;
        ger)
            echo "Deutsch" > $SYNOPKG_TEMP_LOGFILE
        ;;
        hun)
            echo "Magyar" > $SYNOPKG_TEMP_LOGFILE
        ;;
        ita)
            echo "Italiano" > $SYNOPKG_TEMP_LOGFILE
        ;;
        jpn)
            echo ""    > $SYNOPKG_TEMP_LOGFILE
        ;;
        krn)
            echo ""    > $SYNOPKG_TEMP_LOGFILE
        ;;
        nld)
            echo "Nederlands" > $SYNOPKG_TEMP_LOGFILE
        ;;
        nor)
            echo "Norsk" > $SYNOPKG_TEMP_LOGFILE
        ;;
        plk)
            echo "Polski" > $SYNOPKG_TEMP_LOGFILE
        ;;
        ptb)
            echo "Português do Brasil" > $SYNOPKG_TEMP_LOGFILE
        ;;
        ptg)
            echo "Português Europeu" > $SYNOPKG_TEMP_LOGFILE
        ;;
        rus)
            echo "Русский" > $SYNOPKG_TEMP_LOGFILE
        ;;
        spn)
            echo "Español" > $SYNOPKG_TEMP_LOGFILE
        ;;
        sve)
            echo "Svenska" > $SYNOPKG_TEMP_LOGFILE
        ;;
        trk)
            echo "Türkçe" > $SYNOPKG_TEMP_LOGFILE
        ;;
        *)
            echo "English" > $SYNOPKG_TEMP_LOGFILE
        ;;
    esac
```

Please see the "scripts" and "Script Environment Variables" sections for more information.

Otherwise, you can use **/usr/syno/bin/synodsmnotify** to send messages to DSM users. For example, the strings “**title**” and “**messages**” are sent to the “**administrators**” group.

```
synodsmnotify @administrators title messages
```

Create PHP Application

Apache/Nginx and PHP engines which allow you to develop web applications and store files in the web space to build your own website. Hence, these files should be compressed into **package.tgz**. In the **start-stop-status** script, you should link or copy them to the **web** space when starting a package. In DSM, the default web space is the **web** shared folder. You can get the path via **/var/services/web** which is a symbolic link pointing to the actual path in the volume.

In DSM 5.2 or older, the web application works after the administrator enables **Web Station**. You can configure the **install_dep_services** and **start_dep_services** keys with the **apache-web** value in the **INFO** file to make sure **Web Station** is enabled before the package installation. In DSM 6.0 or newer, the web application works after the administrator installs and starts **Web Station** and **PHP** package. You can configure the **install_dep_packages** with **"WebStation:PHP5.6"** value in the **INFO** file to make sure these packages installed and started before the package installation.

You can then provide a URL in order to access your page to **adminurl** (ex:"/myapp/index.html") in **INFO** which will be displayed in Package Center to tell the client which URL to open. When the package stops, this URL should not be accessible. You can remove or unlink the website folder in the web space, or let the web page redirect to an error page.

In addition, it is recommended to add an icon to DSM's main menu so that users can click the icon to launch the application intuitively.

To customize the config settings for Apache, Nginx or PHP, please do not modify any of the config files belonging to DSM. Instead, you should create symbolic links for your config files in the corresponding locations. For Apache/Nginx, link your config to the folder **/etc/httpd/sites-enabled-user/your_package_name.conf**. For PHP, please refer to PHP INI for more detail.

In addition, add the **apache-web** in DSM 5.2 or order, or **nginx** in DSM 6.0 or newer to the **start_stop_restart_services** to make Apache/Nginx reload config files. Otherwise, these symbolic links should be removed along with the package.

Here is an example of apache config:

```
<Directory "/var/services/photo">
        Options MultiViews
        AllowOverride None
        Order allow,deny
        Allow from all
</Directory>
```

Run Scripts When the System Boots

If you would like to run scripts when the system is booting up or shutting down, you can write scripts in **start-stop-status**. This script will be executed with the "start" or "stop" parameter, when the package is enabled. If you want to execute a script during the boot-up or shut-down process, you can put a start-up script in **/usr/local/etc/rc.d/**. Below are the rules for the start-up script:

1. It must contain the suffix "**.sh**". For example, "**myprog.sh**".
2. The permission must be 755.
3. It must contain the options of "start" and "stop". When the system boots up, it will call `myprog.sh start` ; when it shuts down, it will call `myprog.sh stop` .

Locale Support

DSM provides locale support after version 4.3. You do not need to add or remove locale related files on the NAS after this version.

Install Package Related Ports Information into DSM

If your package service uses specific ports for communication (e.g. Surveillance Station uses ports 19997/udp for source port and 19998/udp for destination port), you should prepare a service configuration file for this package to describe which ports will be used. After that, once the user creates firewall rules or port forwarding rules from the build-in application, your package service will also be listed for selection.

Service Configure File Name

The file name should follow the naming convention **SYNOPKG.sc** (ex: **SurveillanceStation.sc**). **SYNOPKG** should be the package name that is specified by the key "**package**" in the **INFO** file, and **sc** means Service Configure file.

Configure Format Template

Please see the following example:

```
[service_name]
title="English title"
desc="English description"
port_forward="yes" or "no"
src.ports="ports/protocols"
dst.ports="ports/protocols"

[service_name2]
…
```

Section/Key Descriptions

Please see the following statements for the strings and keys:

Section/Key	Description	Value	Default Value	DSM Requirement
service_name	*Required* Usually a package only has one unique service name. If your package needs more than one port description, you can define *service_name2, service_name3, …* *Note: service_name cannot be empty and can only include characters "a~z", "A~Z", "0~9", "-", "\", "."*	Unique service name	N/A	4.0-2206
title	*Required* English title which will be shown on field Protocol at firewall build-in selection menu.	English title	N/A	4.0-2206
desc	*Required* English description which will be shown on field Applications at firewall build-in selection menu.	English description	N/A	4.0-2206
port_forward	*Optional* If set to "yes," your package service related ports will be listed when users set port forwarding rule from build-in applications. Otherwise they will not be listed.	"yes" or "no"	"no"	4.0-2206
src.ports	*Optional* If your package service has specified source ports, you can set them in this key. The value should contain at least the port numbers, and a default protocol that is tcp + udp. Ex: 6000,7000:8000/tcp,udp means source ports are 6000, 7000 to 8000, all ports are tcp + udp.	ports/protocols ports: 1~65535 (separated by ',' and use ':' to represent port range) protocols: tcp,udp (separated by ',')	ports: N/A protocols: tcp,udp	4.0-2206
dst.ports	*Required* Each service should have destination ports. The value should contain at least the port numbers, and a default protocol that is tcp + udp. Ex: 6000,7000:8000/tcp,udp means destination ports are 6000, 7000 to 8000, all ports are tcp + udp.	ports/protocols ports: 1~65535 (separated by ',' and use ':' to represent port range) protocols: tcp,udp (separated by ',')	ports: N/A protocols: tcp,udp	4.0-2206

Please see the following example (SurveillanceStation.sc):

```
[ss_findhostd_port]
title="Search Surveillance Station"
desc="Surveillance Station"
port_forward="yes"
src.ports="19997/udp"
dst.ports="19998/udp"
```

After the service configuration file is ready, add the following content to the resource specification file. Please refer to Port Config for more detail.

```
"port-config": {
    "protocol-file": "port_conf/xxdns.sc"
 }
```

Lower Privilege

To reduce security risks, we now provide a framework to run packages with a lower privileged "package user" instead of root. Below is a summary of how to join the framework and what package center does for you:

1. Package developers provide privilege specification to specify what privilege is needed during program execution.
2. During package installation, package center creates corresponding user and group. See Package User & Group for more detail.
3. According to the privilege specification, package center `chown` files under */var/packages/${package}/target*. (The `setuid` and `setgid` bit will be cleared)
4. Package executables are run with privilege (*package user*, *system* or *root*) according to its file owner and group. See Mechanism for more detail.

With this framework, package developer is capable of:

- Configure which executable should be run with what privilege with a simple privilege specification file.
- Resource Acquistion can be used to help maintain some chores that requires root privilege.

Whether to lower the package's privilege and create corresponding user / group is **optional**. The package has to provide privilege specification to join this framework, otherwise the package will still be run with root privilege, and no user / group will be created.

Package User & Group

During package installation, package center uses the package name to create a user and group for the package. And their uid / gid will be the same number, which lies between 100000 and 300000. Failure on user / group creation will cause package installation to abort. If a custom user and group name is preferred, see privilege specification.

User & Group Creation

Newly created user and group will have the below configuration:

```
# /etc/passwd
${package}:x:${uid}:${gid}::/var/packages/${package}/target:/sbin/nologin

# /etc/shadow
${package}:*:10933:0:99999:7:::

# /etc/group
${package}:x:${gid}:
```

UI Behavior

Package users and groups will not appear on most UI settings. Including the following:

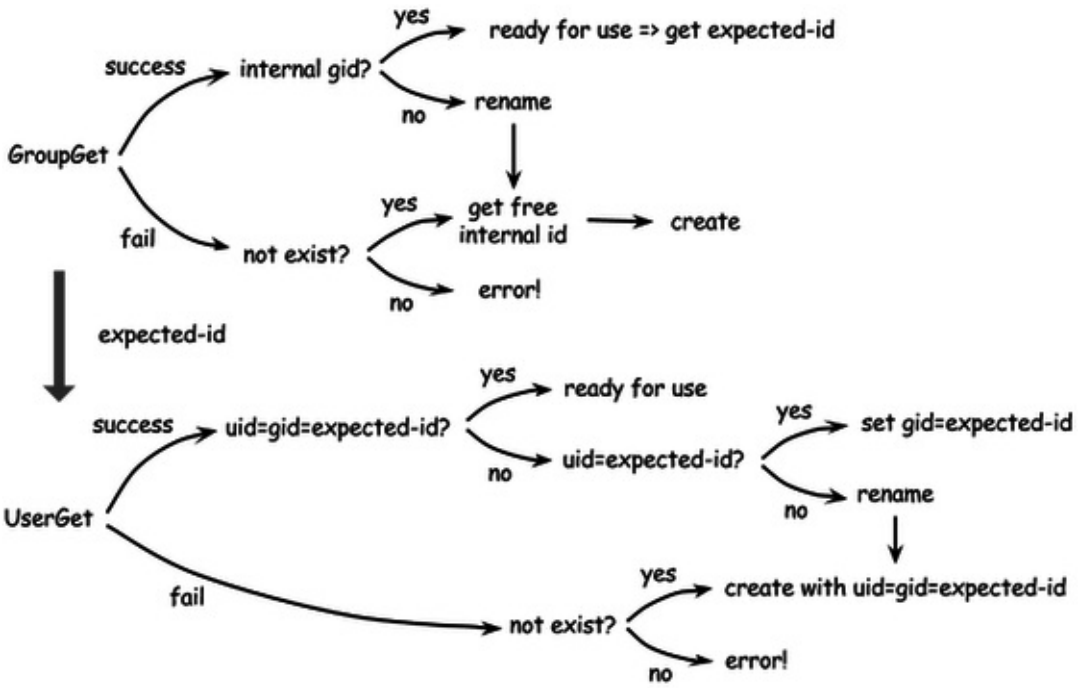
- Application privilege's permission viewer
- FPT's chroot user selector
- File Stations's
  - Change owner
  - Shared Links Manager -> Enable secure sharing

The only exceptions are:

- Control Panel > Shared Folder > Edit > Permission > System internal user
- ACL editor

Creation Policy

When the package user / group name conflicts with a local user / group, it will be renamed by adding a *PKG{time}* postfix at the end. The renaming policy is showed below:


GroupGet
success
internal gid?
yes
ready for use => get expected-id
no
rename
fail
not exist?
yes
get free internal id
create
no
error!
expected-id
UserGet
success
uid=gid=expected-id?
yes
ready for use
no
uid=expected-id?
yes
set gid=expected-id
no
rename
fail
not exist?
yes
create with uid=gid=expected-id
no
error!

Mechanism

A package may contain multiple executables, each of them triggered in different times by the DSM. For example, CGIs will be spawn by the cgi daemon, and control scripts (*preinst*, *start_stop_status*...) will be called by package center. Our framework uses the owner user and group of the executable to decide what user privilege to grant.

Suppose the owner user is ```${package}:

- If the owner group is `system`, then use `setresuid(-1, ${package}, -1), which gives the process ability to change back to` root```.
- Otherwise, use `setresuid(${package}, ${package}, ${package})`, drop its privilege.

A Summory of file owner user / group and granted privilege is shown below:

run-as	owner	euid	ruid
package	${package}:${package}	${package}	${package}
system	${package}:system	${package}	0
root	root:root	0	0

Privilege Specification

During package installation, package files will be `chown` with ${package}:${package}. As mentioned in Mechanism, this results in package executables to be run as ${package}. But some executables might require higher privilege. Here we provide a Json config file called **privilege specification**, located in the SPK's ***conf/privilege***. This **privilege specification** specifies what kind of privilege the executables need.

NOTE For packages that does not support running with lower privileges, simply do not supply this **previlege specification**. Then package center will not apply `chown` on the files.

The content of **privilege specification** is shown below. **defaults** specifies the default privilege and decides how to apply chown on files.. `"username"` and `groupname` are optional, used for custom user / group name. The rest of the key is used to overwrite the `default` setting.

```
{
  "defaults":{
    "run-as": "<run-as>"
  },
  "username": "<username>",
  "groupname": "<groupname>",
  "ctrl-script":[{
    "action": "<action>",
    "run-as": "<run-as>"
  }, ...],
  "executable": [{
    "relpath": "<relpath>",
    "run-as": "<run-as>"
  }, ...],
  "tool": [{
    "relpath": "<relpath>",
    "user": "<user>",
    "group": "<group>",
    "permission": "<mode>"
  }, ...]
}
```

- `<run-as>`
  Can be **`package`** , **`system`** , or **`root`** , determines what privilege will the executable be granted:

run-as	description
`package`	Run as ${package} and drop privilege.
`system`	Run as ${package}, but preserve the ability to grant privilege.
`root`	Run as root.

defaults

Use `run-as` to specify what privilege to be granted, also decides how to chown.

run-as	description
`package`	`chown -hR "${package}:${package}"`
`system`	`chown -hR "${package}:system"`
`root`	Do not chown.

Categories

In here we introduce what categories *privilege specification* supports to overwrite the setting of `defaults`. After applying chown according to `defaults`'s settings, chown again with the caterogy's settings:

- ctrl-script
- executable
- tool

Each category is a key of the **privilege specification** json object, and the corresponding value is a json object array. An object in the array represents the configuration of an executable.

---

ctrl-script

Control scripts (*preinst*, *post_inst*, *start_stop_status*...) are spawned by the Package Center:

```
"ctrl-script": [{
  "action": "<action>",
  "run-as": "<run-as>"
}, ...]
```

Member	Since	Description
`action`	6.0-5891	String, can only be one of "preinst", "postinst", "preuninst", "postuninst", "preupgrade", "postupgrade", "start", "stop", "status", "prestart", "prestop", or "log"
`run-as`	6.0-5891	See defaults.

executable

Executables **directly** spawned by the DSM framework belongs to this category:

```
"executable": [{
  "relpath": "<relpath>",
  "run-as": "<run-as>"
}, ...]
```

Member	Since	Description
`relpath`	6.0-5891	String, the file's relative path under */var/packages/${package}/target/*.
`run-as`	6.0-5891	See defaults.

tool

Executables **indirectly** called by the DSM framework spawn belongs to this category (For example, command line tool called by a CGI):

```
"tool": [{
  "relpath": "<relpath>",
  "user": "<user>",
  "group": "<group>",
  "permission": "<mode>"
}, ...]
```

Member	Since	Description
`relpath`	6.0-5891	String, the file's relative path under */var/packages/${package}/target/*.
`user`	6.0-5891	String, file's owner user, can only be "package" or "root".
`group`	6.0-5891	String, file's owner group, can only be "package" or "root".
`permission`	6.0-5891	4 digit number to set file permission, for example: 4750

Resource Acquisition

During package installation and start-up, we often need to install config files (e.g. php.ini, apache.conf) or register services (e.g. port config, data share) onto the DSM. After setting package to run with lower privilege, you might lose the ability to do these job in the control scripts. Instead of writing these jobs in the installation scripts (*preinst*, *postinst*...), we provide a framework to simplify the process.

The config files and services that needs to be installed and registered are called **resource**. Each **resource** has a corresponding **worker**, which is responsible for installing the **resource** onto the DSM. Package developers just need to follow a predefined syntax and provide the **resource** **specification**, Package Center will then call the correspoding **worker** to finish the job.

NOTE If a package is using this framework, please remember to adjust the *INFO* file's `firmware` to meet the **worker**'s requirement

Resource Specification

The json file *resource* is used to specify a package's required resources, and it should be placed in the *conf* folder. After package installation, this file will be located at */var/packages/${package}/conf/resource*. The content of the file should be something like:

```
{
  "<resource-id>": <specification>,
  ...
}
```

- `<resource-id>`
  String, represents the resource's ID.
- `<specification>`
  Json object, describes the resource requirements.

Take the below */usr/local linker* worker for example:

```
{
  "usr-local-linker": {
    "lib": ["lib/foo"],
    "bin": ["bin/foo"],
    "etc": ["etc/foo"],
  }
}
```

`"usr-local-linker"` is the resource ID and the correponding Json object is the specification, which specifies what files should be installed into the DSM.

Data from Wizard

As mentioned in *WIZARD_UIFILES*, installation scripts can optain user's input from the UI wizard, some **workers** are also capable of doing this. In the *resource* file, variables surrounded by `{{}}` will be replaced by the user input.

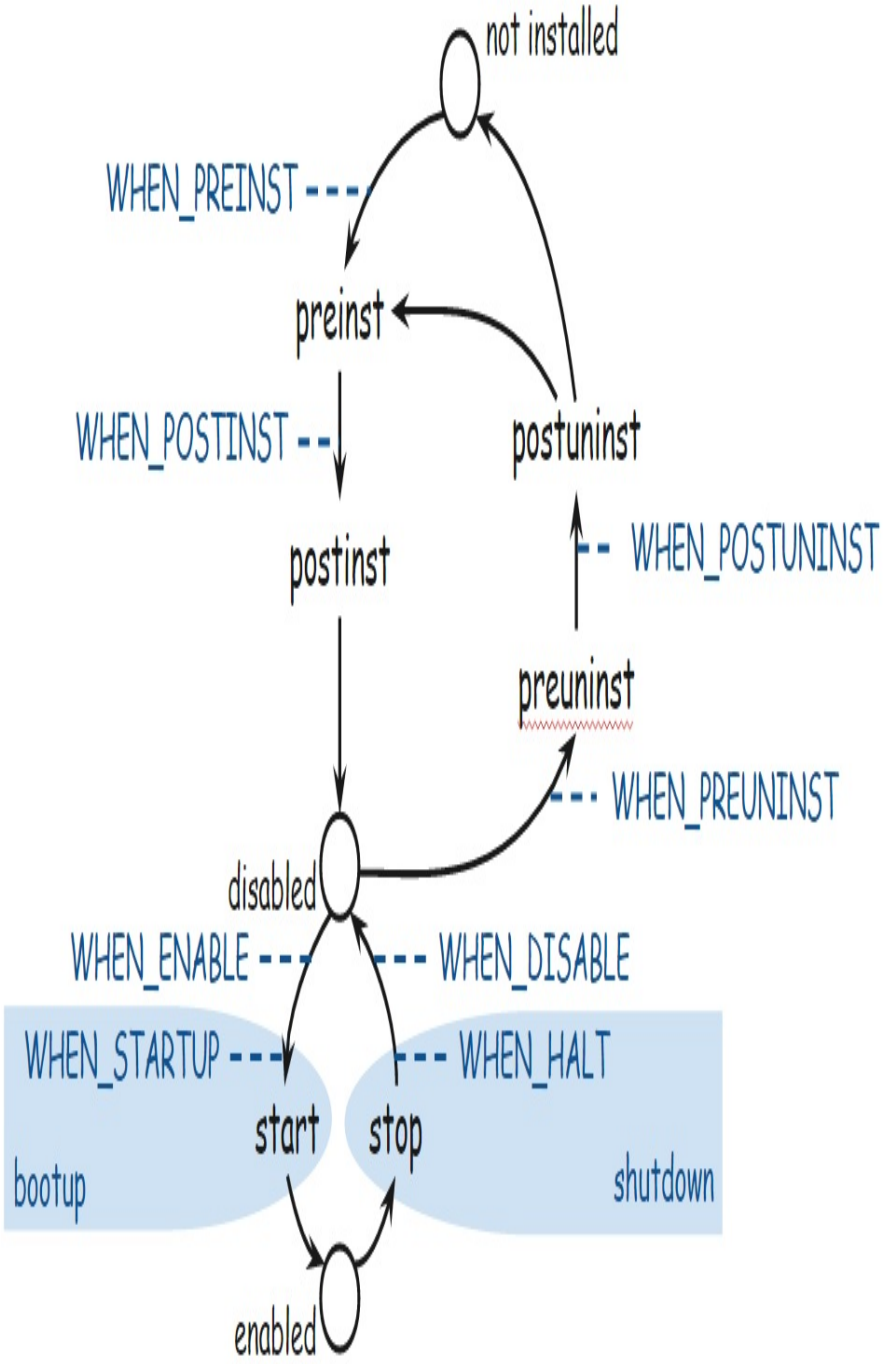
Take the below *Data Share* worker for example:

```
"data-share": {
  "name": "{{pkgwizard_share_name}}",
  "permission": {
    "ro": ["admin"]
  }
}
```

`{{pkgwizard_share_name}}` will be replaced by *WIZARD_UIFILES/install_uifile*'s or *WIZARD_UIFILES/upgrade_uifiles*'s `pkgwizard_share_name`

Timing

Every worker acquires resources at certain timings and holds it during an interval. For example, */usr/local linker* holds the resource during the interval `FROM_ENABLE_TO_DISABLE`, which means it acquires resource at `WHEN_ENABLE` and releases it at `WHEN_DISABLE`. The timings are listed and explained below:



timing	descrioption	when Failure
`WHEN_PREINST`	before *preinst*	abort installation, rollback, show alert message on UI
`WHEN_POSTINST`	before *postinst*	finish installation, show alert message on UI
`WHEN_ENABLE`	before `WHEN_STARTUP`, **won't process during bootup**	abort startup, rollback, show alert message on UI
`WHEN_STARTUP`	before *start*	abort startup, rollback, show alert message on UI
`WHEN_PREUNINST`	after *preuninst*	finish uninstallation, show alert message on UI
`WHEN_POSTUNINST`	before *postuninst*	finish uninstallation, show alert message on UI
`WHEN_DISABLE`	after `WHEN_HALT`, **won't process during shutdown**	ignore
`WHEN_HALT`	after *stop*	ignore

NOTE To let the package itself decide whether uninstallation should continue or not, `WHEN_PREUNINST` is processed after the `preuninst` script.

Config Update

Some workers support config update after package installation. */usr/syno/sbin/synopkgheler* should be used to accomplish the job. Below are the steps to update the config:

1. Update the content of config file under */var/packages/${package}/target/*.
2. Execute the command `/usr/syno/sbin/synopkghelper update <package> <resource-id>` to trigger the corresponding worker to update the config.

For example, suppose a package allows the user to edit its listening port and needs to trigger the Port config worker for update:

1. Provide the user some UI to input the port number.
2. After receiving the new port number, update the config file under */var/packages/${package/target/*.
3. Execute the command `/usr/syno/sbin/synopkghelper update ${package} protocol_file`, the worker will re-read the config file and reload firewall and portforwarding.

NOTE Not all worker supports config update, please refer to the *Updatable* section of each worker.

Available Workers

As mentioned in the section Resource Acquisition, a worker is needed for resource management. Given a Resource Specification configuration file, the worker will install\/uninstall the described resource at certain time. This section describes the available workers on the DSM.

/usr/local linker

Description

Package's executables and library files should be installed to */usr/local*. This worker link / unlink files to ***/usr/local/{bin,lib,etc}*** during package start / stop.

- `Acquire()` : Create symbolic links under */usr/local/{bin,lib,etc}/* that points to files in */var/packages/${package}/target/*.
  - Files not found under */var/packages/${package}/target/* will be ignored.
  - If the target file already exists in */usr/local/{bin,lib,etc}*, it will be `unlink()` first.
  - Failure on any file link results in this worker to abort and triggers rollback.
- `Release()` : Delete the links under */usr/local/{bin,lib,etc}/*.
  - Ignore files that are not found.
  - Ignore `unlink()` failure.

Provider

DSM

Timing

`FROM_ENABLE_TO_DISABLE`

Environment Variables

None

Updatable

No

Syntax

```
"usr-local-linker": {
  "bin" ["<relpath>", ...],
  "lib" ["<relpath>", ...],
  "etc" ["<relpath>", ...]
}
```

Member	Since	Description
`bin`	6.0-5941	String array, list of files to be linked under */usr/local/bin/*.
`lib`	6.0-5941	String array, list of files to be linked under */usr/local/lib/*.
`etc`	6.0-5941	String array, list of files to be linked under */usr/local/etc/*.
`relpath`	6.0-5941	String, target file's relative path under */var/packages/${package}/target/*.

Example

```
"usr-local-linker": {
  "bin": ["usr/bin/a2p", "usr/bin/perl"],
  "lib": ["lib/perl5"]
}
```

The above specifications generates the following symbolic links for the Perl package:

```
root@DS $ ls -l /usr/local/{bin,lib,etc}
/usr/local/bin/:
total 0
lrwxrwxrwx 1 root root   30 Aug 13 06:32 a2p -> /var/packages/Perl/target/usr/bin/a2p
lrwxrwxrwx 1 root root   31 Aug 13 06:32 perl -> /var/packages/Perl/target/usr/bin/perl

/usr/local/lib/:
total 0
lrwxrwxrwx 1 root root   28 Aug 13 06:32 perl5 -> /var/packages/Perl/target/lib/perl5

/usr/local/etc/:
total 0
```

Apache 2.2 Config

Description

Packages can carry sites-enabled/*.conf files for Apache HTTP Server 2.2. This worker installs / uninstalls these config files during package start / stop.

- `Acquire()` : Copy the conf files to */usr/local/etc/httpd/sites-enabled/*. Then reload Apache 2.2.
  - The files should have .conf extension, otherwise it will be ignored
  - Files will be prefixed by ${package}.
  - Existing files will be `unlink()` first.
  - Failure on any file copy results in this worker to abort and triggers rollback.
- `Release()` : Delete previously created links
  - Ignore files that are not found.
  - Ignore `unlink()` failure.

Provider

WebStation

Timing

`FROM_ENABLE_TO_DISABLE`

Environment Variables

None

Updatable

No

Syntax

```
"apache22": {
    "sites-enabled": [{
        "relpath": "<conf-relpath>",
    }, ...]
}
```

Member	Since	Description
`sites-enabled`	WebStation-1.0-0049	Object array, list of conf files to install.
`relpath`	WebStation-1.0-0049	Target file's relative path under */var/packages/${package}/target/*.

Example

```
{
    "apache22": {
        "sites-enabled": [{
            "relpath": "synology_added/test_1.conf"
        }, {
            "relpath": "synology_added/test_2.conf"
        }, {
            "relpath": "synology_added/test_3.conf"
        }]
    }
}
```

Data Share

Description

This worker creates shared folder and set its permission during package startup. The share name can be hard-coded in the specification or given by user input from the UI wizard. The shared folder will not be removed after package uninstallation, since it might delete the user's personal data as well.

- `Acquire()` : Create shared folder and set its permission.
  - If the shared folder already exists, skip share creation and set the permission.
- `Release()` : Does nothing.

Provider

DSM

Timing

`FROM_ENABLE_TO_POSTUNINST`

Environment Variables

None

Updatable

No

Syntax

```
"data-share": {
  "shares": [{
    "name": "<share-name>",
    "permission": {
      "ro": ["<user-name>", ...],
      "rw": ["<user-name>", ...]
    },
    "once": "<once>"
  }, ...]
}
```

Member	Since	Description
`shares`	6.0-5914	Object array, array of shares to create
`name`	6.0-5914	String, name of the share, can be obtained from the UI wizard
`permission`	6.0-5914	Json object, permission of the share. (optional)
`ro`	6.0-5914	String arrayusers to be assigned with read-only permission.
`rw`	6.0-5914	String arrayusers to be assigned with read / write permission.
`once`	6.0-5914	Boolean, only try to create share on package's first start. (optional, default = `false`)

Example

The following specification creates a share *music*, and gives the user *AudioStation* read-only permission. Since `once` defaults to `false`, the above procedure is ran every time the package starts.

```
"data-share": {
  "shares": [{
    "name": "music",
    "permission": {
      "ro": ["AudioStation"]
    }
  }]
}
```

The following specification reads the share name from *WIZARD_UIFILES/install_uifile*'s `pkgwizard_share_name`, and gives the user *admin* read-only permission.

```
"data-share": {
  "shares": [{
    "name": "{{pkgwizard_share_name}}",
    "permission": {
      "ro": ["admin"]
    }
  }]
}
```

Index DB

Description

Index / unindex package help and app index during package start / stop.

For detailed description on package app index and help index, please refer to Integegrate Help Document into DSM Help.

- `Acquire()` : Index package help and app content.
- `Release()` : Un-index package help and app content.

Provider

DSM

Timing

`FROM_ENABLE_TO_DISABLE`

Environment Variables

None

Updatable

No

Syntax

```
"indexdb": {
    "app-index"  : {
        "conf-relpath": "<conf relpath>",
        "db-relpath": "<app db relpath>"
    },
    "help-index": {
        "conf-relpath": "<conf relpath>",
        "db-relpath": "<help db relpath>"
    }
}
```

Member	Since	Description
`app-index`	6.0-5924	Object, app index info.
`help-index`	6.0-5924	Object, help index info.
`conf-relpath`	6.0-5924	String, config file's relative path under */var/packages/${package}/target/*.
`db-relpath`	6.0-5924	String, db folder's relative path under */var/packages/${package}/target/*.

Example

```
"indexdb": {
    "app-index"  : {
        "conf-relpath": "app/index.conf",
        "db-relpath": "indexdb/appindexdb"
    },
    "help-index": {
        "conf-relpath": "app/helptoc.conf",
        "db-relpath": "indexdb/helpindexdb"
    }
}
```

Maria DB

Description

Create / delete database and db-user during package install / uninstall. The database name can be hard-coded, read from config file or given by the user from UI wizard. Package developer can decide if creating a corresponding db-user is needed or not.

- `Acquire()` : Create database and db-user accroding to *resource specification*.
  - **database**: *resource specification* can specify what action to take if a database with the same name already exists:
    1. `drop` : Delete the existing database.
    2. `skip` : Ignore database creation, keep the existing database and conitinue installation.
    3. `error` : Return error and rollback. (Default: `error`)
  - **db-user**: Create db-user and grant access to the database according to the *resource specification*. The default db-user to create is `'${package}'@'localhost'`
- `Release()` : Delete database and db-user accroding to the *resource specification*. Default is to keep the database and db-user.
- During package upgrade, MariaDB worker provides the get_key_value method to obtain previously create database's name, and there will be no create / delete for database and db-user

Provider

MariaDB

Timing

`FROM_PREINST_TO_PREUNINST`

Environment Variables

Variable	Since	Description
`SYNOPKG_DB_USER_RAND_PW`	6.0-5920	The random password generated during datebase user creation.

Updatable

No

Syntax

(`(*)` required)

```
"mysql-db": {
    "info": {
        "db-name": "<db name>",     (*)
        "conf": "<conf>",
        "key": "<key>"
    },
    "root-pw": "<db password>", (*)
    "create-db" : {
        "char-set": "<character-set>",
        "collate": "<collate>",
        "db-collision": "drop" | "skip" | "error"
    },
    "grant-user": {
        "user-name" : "<db username>",
        "host"      : "<db user host>",
        "user-pw"   : "<user password>",
        "rand-pw"   : true | false
    },
    "drop-db-uninst": true | false,
    "drop-user-uninst": true | false
}
```

Member	Since	Description
`info`	5.5.47-0062	Object, info of db-name. The priority of retrieving database's name is `db-name` > `conf`. Which means, if *db-name* is given, *conf* and *key* will be ignored.
`db-name`	5.5.47-0062	String, database name, can be given by UI wizard.
`conf`	5.5.47-0062	String, file containing the database name's key-value-pair.
`key`	5.5.47-0062	String, the *key* in `conf` file to look for db-name.
`root-pw`	5.5.47-0062	String, root password of MariaDB.
`create-db`	5.5.47-0062	Object, info of database. If does not exist, database will not be created during `Acquire()`.
`character-set`	5.5.47-0062	String, database's *CHARACTER SET*. (default = *utf8*)
`collate`	5.5.47-0062	String, database's *COLLATE*. (default = *utf8_unicode_ci*)
`db-collision`	5.5.47-0062	String, action to take if the database exists. Can be `drop` / `skip` / `error`.
`grant-user`	5.5.47-0062	Object, info of db-user. If does not exist, db-user will not be created during `Acquire()`.
`user-name`	5.5.47-0062	String, db-user name. (default = `${package}`)
`host`	5.5.47-0062	String, db-user's host. (defaults = `localhost`)
`user-pw`	5.5.47-0062	String, db-user's password. If empty or null, db-user's password will not be set. Overrides existing user's password.
`rand-pw`	5.5.47-0062	Boolean, whether to generate a random password. If set to `true`, the db-user will be given a random password and be passed to environment variable `SYNOPKG_DB_USER_RAND_PW`. If `user-pw` exists, this value will be ignored.
`drop-db-uninst`	5.5.47-0062	Boolean, whether to delete database during `Release()`. Can be given by UI wizard. (defaults = `false`)
`drop-user-uninst`	5.5.47-0062	Boolean, whether to delete db-user during `Release()`. Can be given by UI wizard. (defaults = `false`)

Example

```
"mysql-db": {
    "info": {
        "db-name": "wordpressblog"
    },
    "root-pw": "{{pkgwizard_mysql_password}}",
    "create-db" : {
        "db-collision": "skip"
    },
    "grant-user": {
        "user-name" : "wordpress"
    },
    "drop-db-uninst": "{{pkgwizard_remove_mysql}}",
    "drop-user-uninst": "{{pkgwizard_remove_mysql}}"
}
```

PHP INI

Description

Packages can carry custom php.ini and fpm.conf files. This worker installs / uninstalls these config files during package start / stop.

- `Acquire()` : Copy the php.ini and fpm.conf files to */usr/local/etc/php56/conf.d/* and */usr/local/etc/php56/fpm.d/*. Then reload php56-fpm.
  - *php.ini* / *fpm.conf* files should have .ini / .conf extension, otherwise it will be ignored
  - Files will be prefixed by ${package}.
  - Existing files will be `unlink()` first.
  - Failure on any file copy results in this worker to abort and triggers rollback.
- `Release()` : Delete previously created links
  - Ignore files that are not found.
  - Ignore `unlink()` failure.

Provider

PHP5.6

Timing

`FROM_ENABLE_TO_DISABLE`

Environment Variables

None

Updatable

No

Syntax

```
"php": {
     "php-ini": [{
         "relpath": "<ini-relpath>",
     }, ...],
     "fpm-conf": [{
         "relpath": "<conf-relpath>",
     }, ...]
 }
```

Member	Since	Description
`php-ini`	PHP5.6-5.6.17-0020	Object array, list of php.ini files to install.
`fpm-conf`	PHP5.6-5.6.17-0020	Object array, list of fpm.conf files to install.
`relpath`	PHP5.6-5.6.17-0020	Target file's relative path under */var/packages/${package}/target/*.

Example

```
{
    "php": {
        "php-ini": [{
            "relpath": "synology_added/etc/php/conf.d/test_1.ini"
        }, {
            "relpath": "synology_added/etc/php/conf.d/test_2.ini"
        }, {
            "relpath": "synology_added/etc/php/conf.d/test_3.ini"
        }],
        "fpm-conf": [{
            "relpath": "synology_added/etc/php/fpm.d/test_1.conf"
        }, {
            "relpath": "synology_added/etc/php/fpm.d/test_2.conf"
        }, {
            "relpath": "synology_added/etc/php/fpm.d/test_3.conf"
        }]
    }
}
```

Port Config

Description

Install / uninstall service port config file during package install / uninstall.

For detailed description on what is and how to write a port config file, please refer to Install Package Related Ports Information into DSM.

- `Acquire()` : copy the .sc file to */usr/local/etc/service.d/*
  - If the destination file exists, skip file copy.
- `Release()` : remove the .sc file and reload the firewall and portforward.
- `Update()` : update the .sc file and reload firewall and portforward.

Timing

`FROM_POSTINST_TO_POSTUNINST`

Environment Variables

None

Updatable

Yes, please refer to Config Update on how to trigger update.

Syntax

```
"port-config": {
    "protocol-file": <protocol_file>
 }
```

Member	Since	Description
`protocol_file`	6.0-5936	.sc file's relative path under */var/package/{$package}/target/*

Example

```
"port-config": {
    "protocol-file": "port_conf/xxdns.sc"
 }
```

Syslog Config

Description

Install / uninstall the syslog-ng and logrotate config file during package start / stop.

Please refer to syslog-ng on how to write the syslog-ng's config file.

- `Acquire()` : Copy patterndb / logratoate to */usr/local/etc/syslog-ng/patterndb.d/* / */usr/local/etc/logrotate.d/*. Then reload syslog-ng.
  - If file exists, `unlink()` it first.
  - Failure on any file copy results in this worker to abort and triggers rollback.
- `Release()` : Delete the config files and reload syslog-ng.
  - Ignore `unlink()` failure.

Provider

DSM

Timing

`FROM_ENABLE_TO_DISABLE`

Environment Variables

None

Updatable

No

Syntax

```
"syslog-config": {
  "patterndb-relpath": "<relpath>",
  "logrotate-relpath": "<relpath>"
}
```

Member	Since	Description
`patterndb-relpath`	6.0-7145	String, syslog-ng's config file's relative path under */var/packages/${package}/target/*, ignore this if the log is not generated by syslog-ng (optional)
`logrotate-relpath`	6.0-5911	String, logrotate's config file's relative path under */var/packages/${package}/target/*, ignore this if log is saved to database (optional)

Example

```
"syslog-config": {
  "patterndb-relpath": "etc/syslog-ng.conf",
  "logrotate-relpath": "etc/logrotate.conf"
}
```

Publish Synology Packages

Get Started with Publishing

To publish in Synology Package Center requires a few simple steps. Here is how to do it:

1. Acquire a Synology Checkout Merchant Account via a Synology specialist (for more details please contact `package@synology.com`).
2. Read and accept the Developer Distribution Agreement. Note that packages that you publish on Package Center must comply with the Terms of Service in Package Center.

Please note that the package quality directly influences the long-term success of your package in terms of installation, online reviews, engagement, and user retention.

Submitting the Package for Approval

Before you publish your package in Package Center and distribute it to users, you need to get the package (the SPK file) ready, test it, and prepare your promotion materials if needed. Please see the checklist below before submitting your package to us.

Confirm Package Size

The overall size of your package can affect its design and how you publish it in Package Center. Currently, the maximum size for a SPK file published on Package Center is **100MB**.

Free or Paid Package

In Package Center, you can publish free or paid packages. Free packages can be downloaded by any user in Package Center. Paid apps can be downloaded only by users who have a registered Synology Account.

Deciding whether your package will be free or paid is important because **free packages must remain free**.

- Once your package is published as a free one, you cannot change it to a paid package.
- If you publish your package as a paid one, you can change it to free at any time (but cannot be changed back to paid).
- If your package is paid, you need to set up a Synology Checkout Merchant Account before the package can be published.
- For queries about the Synology Checkout Merchant Account, you can contact `package@synology.com` .

Set a Price for Your Package

If you have a paid package, Synology lets you set prices for your package only in US currency for users in markets around the world. Before you publish, consider how you will price your package and what your price will be in various currencies.

Prepare Screenshots

When you publish in Package Center, you must supply a variety of high-quality screen-shots to showcase your package or brand. After you publish, they will appear on your package details page, or elsewhere. These screen-shots are a key part of a successful package details page that will attract and engage users. Therefore, you may also consider hiring a professional to produce them for you.

Submit Your Package

When you are ready to publish, send an email to a Synology specialist `package@synology.com` to make your submission.

Make sure that:

- You have applied through Synology Dev Center and become an authorized developer.
- Your package is the right version.
- You provide a download link for your package.
- You provide a package description with what it does.
- You provide a change log with what was updated in this version.
- Package pricing is set to be free or paid (for the first time submission).
- The link to your website and the support email address is correct.
- You have acknowledged that your package meets the Developer Distribution Agreement and also the Terms of Service from Package Center.

Responding to User Issues

After you publish a package, it it crucial for you to offer support to your customers. Prompt and courteous support can provide a better experience for users, which can result in more downloads and more positive online reviews for your packages. Users are more likely to be more engaged with your package and recommend it if you are responsive to their needs and feedback.

There are many ways that you can keep in touch with users and offer them support. The most common way is to provide a support email address in your package details page. You can also provide support in other ways, such as a forum or a mailing list. The Synology technical support team provides user support for downloading, installing and payments issues, but issues that fall outside of these topics will fall under your domain. Examples of issues you can support include: feature requests, questions about using the app and questions about compatibility settings.

After publishing, please plan to:

- Provide a link to your support resources and set up any other support outlets such as a forum.
- Provide an appropriate support email address on your package detail page and respond to users when they email you.
- Acknowledge and fix issues with your package. It helps to be transparent and list known issues on your package details page regularly.
- Publish updates frequently, without sacrificing quality or annoying users with too-frequent updates.
- With each update, make sure you provide a summary of what is new. Users will read it and appreciate that you are serious about improving the quality of your package.

Appendix A: Platform and Arch Value Mapping Table

The architecture of the NAS is developed upon various platforms on which your package is designed and needs to be addressed in the **INFO** file in the package.

In the below table, you will find the string value corresponding to the platform in question. For example, if the platform of your NAS is Marvell Kirkwood, 88F6281, the value that should to be provided as a pair of the arch key is 88f6281.

Please check the platforms of the NAS to be supported and refer to the table below for their corresponding string values:

Platform Name	Arch Value
Marvell Kirkwood, 88F6281	88f6281
Marvell Kirkwood, 88F6282	88f6282
Intel Atom D410/D510 (Pineview)	x86
Intel Atom D2700 (Cedarview)	cedarview
Intel SandyBridge, Intel IvyBridge, Intel Haswell	bromolow
Marvell Armada 370	armada370
Marvell Armada 375	armada375
Marvell Armada XP	armadaxp
Annapurnalabs, Alpine	alpine/alpine4k
Mindspeed, Comcerto, C2000	comcerto2k
Intel Atom CE SoC	evansport
No platform dependency	noarch

Revision History

This table describes the changes to the Synology DSM 3rd Party Apps Developer Guide.

Date	Note
2008/06/16	1. Original release date of document.
2009/02/09	1. Added Freescale 8533 tool chain information.
2009/03/09	1. Added Marvell 6281 tool chain information. 2. Added Desktop Icon chapter.
2010/05/20	1. Added related information for 10 models. 2. Changed all DSM 2.0 & 2.1 to DSM 2.3. 3. New screenshots for desktop icon & DSM application. 4. Changed configuration files and tool chains. 5. DS1010+ uses Intel Atom D510.
2010/05/28	1. Revised Tool Chain description 2. Revised 88f5182-config to 88f5281-config in Kernel Module. 3. Revised path description of application.cfg / desktop.cfg.
2010/11/29	1. Renamed the document to “Synology DiskStation Manager 3rd-Party Apps Developer Guide”. 2. Added DSM 3.0 Integration section.
2011/10/31	1. Added Synology package creation section. 2. Updated DSM tool chain information.
2012/03/20	Added the guideline to application storage, web application running on Apache and created a shared folder.
2012/09/19	Added Freescale QorIQ tool chain information.
2013/03/11	1. General update for DSM 4.2 release. 2. Added section regarding payment framework. 3. Updated formatting.
2013/05/06	Added Marvell Armada 370 tool chain information.
2013/05/29	Added Marvell Armada XP and Evansport tool chain information.
2013/06/05	Updated QorIQ, Armada XP, Armada 370, Evansport $CC variable.
2013/07/09	1. Updated install_dep_services and start_dep_services in INFO section. 2. Added Locale Support.
2013/07/25	1. Updated Linux kernel version. 2. Added CSRF protection in DSM Web Authentication section.
2013/08/29	General update on tool chain, tool kit, and GPL kernel source for DSM 4.3 release.
2014/02/25	1. Updated keys in INFO section for DSM 5.0. 2. Added DSM 5.0 support in platform chart. 3. Updated Create PHP Application with DSM 5.0 change. 4. Added Create User Account and Share Permission. 5. Minor corrections.
2014/03/26	1. Added Appendix for Platform and Arch string mapping table. 2. Added compilation instruction for DSM 5.0 build.
2014/08/17	1. Added code sign mechanism for packages.
2014/10/29	1. Added Chapter: Quick Start Guide. 2. Removed the duplicated content.
2015/09/10	1. Aded Marvell Armada 375ANNAPURNALABS, Alpine AL212/AL314/AL514 and MINDSPEED, Comcerto, C2000 tool chain information.
2016/01/14	1. Content revised. 2. Changed to format to Gitbook.

Compile Applications

The Synology NAS employs embedded SoC or x86-based CPUs, implementing several platforms -- such as ARM and PowerPC -- on a variety of Synology NAS models. In order to run 3rd-party applications on the Synology NAS, it is necessary to compile applications into an executable format for the corresponding platform.

The table below lists the CPU, architecture, Endianness, and Linux kernel version of each Synology NAS model. This information will help you determine which DSM tool chain (please refer to the “Download DSM Tool Chain” section) to download for each model.

Please refer to What kind of CPU does my NAS have ***Need to update the hyperlink***for a complete model list.

Model (To name a few)	CPU	Arch	Endianness	Linux
DS112j, DS112, DS112+, DS411slim, DS213, DS212j	Marvell 6281 Marvell 6282	ARM	Little Endian	2.6.32
DS213j	Marvell Armada 370	ARM	Little Endian	3.2.40
DS214, DS214+	Marvell Armada XP	ARM	Little Endian	3.2.40
DS215+, DS416	Annapurnalabs, Alpine AL212	ARM	Little Endian	3.2.40
DS715, DS1515	Annapurnalabs, Alpine AL314	ARM	Little Endian	3.2.40
DS2015xs	Annapurnalabs, Alpine AL514	ARM	Little Endian	3.2.40
DS115, DS215j	Marvell Armada 375	ARM	Little Endian	3.2.40
DS414j	Mindspeed, Comcerto, C2000	ARM	Little Endian	3.2.40
DS712+, DS2412+, RS2212+, DS1512+, DS1812+, DS412+, RS812+	Intel Atom	Intel x86	Little Endian	3.2.40
DS3612xs, RS3412xs, RS3412RPxs	Intel Core i3	Intel x86	Little Endian	3.2.40
DS214play	Intel SoC	Intel x86	Little Endian	3.2.40
DS213+, DS413	Freescale QorIQ P1022	PowerPC	Big Endian	2.6.32

To compile an application for the Synology NAS, a compiler that runs on Linux PC is required in order to generate an executable file for the Synology NAS. This compiling procedure is called *cross-compiling*, and the set of compiling tools (compiler, linker, etc) used to compile the application is called a *tool chain.*

Download DSM Tool Chain

To download the DSM tool chain, please go to SourceForge. The table below shows the file name of tool chains for NAS with different CPUs:

CPU	Tool Chain	Linux
Marvell 6281	Marvell 88F628x Linux 2.6.32	2.6.32
Marvell Armada 370	Marvell Armada 370 Linux 3.2.40 3	3.2.40
Marvell Armada XP Marv	Marvell Armada xp Linux 3.2.40	3.2.40
Marvell Armada 375	Marvell Armada 370 Linux 3.2.40	3.2.40
Annapurnalabs, Alpine AL212/AL314/AL514	Annapurnalabs, Alpine Linux 3.2.40	3.2.40
Mindspeed, Comcerto, C2000	Mindspeed, Comcerto, C2000 Linux 3.2.40	3.2.40
Freescale QorIQ (P1022)	PowerPC QorIQ Linux 2.6.32	2.6.32
Intel Atom	Intel x86 Linux 3.2.40 (Pineview) Intel x86 Linux 3.2.40 (Cedarview)	3.2.40
Intel Core i3	Intel x86 Linux 3.2.40 (Bromolow)	3.2.40
Intel SoC	Intel x86 Linux 3.2.40 (Evansport)	3.2.40

If you are not sure about which tool chain you need, please execute the following command on your Synology NAS.

```
# uname -a
Linux myds 3.2.40 #3503 SMP Thu Mar 21 15:17:31 CST 2013 x86_64
GNU/Linux synology_x86_712+
```

The last “synology_x64_712+” tells you which tool chain is appropriate. For examples, x86 means you need the tool chain for Pineview.

After you download the DSM tool chain, extract it to where you want it on your computer. For the following instructions we will extract to **/usr/local/** as an example. You can extract the tool chain by using the following command:

```
# tar zxpf gcc343_glibc232_88f5281.tgz -C /usr/local/
```

Please make sure the tool chain is located in the directory **/usr/local** on your computer to ensure proper integration.

Compile

You can start compiling an application called “minimalPkg.c”, for example, that looks like this:

```
#include <sys/sysinfo.h>

int main()
{
    struct sysinfo info;
    int ret;
    ret = sysinfo(&info);
    if (ret != 0) {
        printf("Failed to get system information.\n");
        return -1;
    }
    printf("Total RAM: %u\n", info.totalram);
    printf("Free RAM: %u\n", info.freeram);
    return 0;
}
```

To compile the application, run the following command:

```
/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linux-gnueabigcc minimalPkg.c –o sysinfo
```

You can also write a Makefile for it:

```
EXEC= sysinfo
OBJS= sysinfo.o

CC= /usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-gcc
LD= /usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-ld
CFLAGS += -I/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/include
LDFLAGS += -L/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/lib

all: $(EXEC)

$(EXEC): $(OBJS)
    $(CC) $(CFLAGS) $(OBJS) -o $@ $(LDFLAGS)

clean:
    rm -rf *.o $(PROG) *.core
```

Compile Open Source Projects

To compile an application on most open source projects, you will be asked to execute the following three steps:

1. `configure`
2. `make`
3. `make install`

The configure script basically consists of many lines which are used to check details about the machine on where the software is going to be installed. The script will check for a lot of dependencies on your system. When you run the configure script, you will see a lot of output on the screen, each being some sort of question with a respective yes/no reply. If there are any major requirements missing on your system, the configure script will exit and you will not be able to proceed with the installation until you meet all the requirements. In most cases, compile applications on some particular target machines will require you to modify the configure script manually to provide the correct values.

When running the configure script to configure software packages for cross-compiling, you will need to specify the `CC` , `LD` , `RANLIB` , `CFLAGS` , `LDFLAGS` , `host` , `target` , and `build` , etc. Some examples are given below.

For PowerPC QorIQ platform in DSM 5.0:

```
env CC=/usr/local/powerpc-none-linux-gnuspe/bin/powerpc-none-linuxgnuspe-gcc \
    LD=/usr/local/powerpc-none-linux-gnuspe/bin/powerpc-none-linuxgnuspe-ld \
    RANLIB=/usr/local/powerpc-none-linux-gnuspe/bin/powerpc-none-linuxgnuspe-ranlib \
    CFLAGS="-I/usr/local/powerpc-none-linux-gnuspe/include -mcpu=8548 -mhard-float -mfloat-gprs=double" \
    LDFLAGS="-L/usr/local/powerpc-none-linux-gnuspe/lib" \
./configure \
    --host=powerpc-unknown-linux \
    --target=powerpc-unknown-linux \
    --build=i686-pc-linux \
    --prefix=/usr/local
```

For Marvell 6281 platform in DSM 5.0:

```
env CC=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-gcc \
    LD=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-ld \
    RANLIB=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-ranlib \
    CFLAGS="-I/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/include" \
    LDFLAGS="-L/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/lib" \
./configure \
    --host=armle-unknown-linux \
    --target=armle-unknown-linux \
    --build=i686-pc-linux \
    --prefix=/usr/local
```

For Marvell Armada 370 platform in DSM 5.0:

```
env CC=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-gcc \
    LD=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-ld \
    RANLIB=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-ranlib \
    CFLAGS="-I/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/include -mhard-float -mfpu=vfpv3-d16" \
    LDFLAGS="-L/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/lib" \
./configure \
    --host=armle-unknown-linux \
    --target=armle-unknown-linux \
    --build=i686-pc-linux" \
    --prefix=/usr/local
```

For Marvell Armada 375 platform in DSM 5.1:

```
env CC=/usr/local/armv7-marvell-linux-gnueabi-hard/bin/arm-marvelllinux-gnueabi-ccache-gcc \
    LD=/usr/local/armv7-marvell-linux-gnueabi-hard/bin/arm-marvell-linuxgnueabi-ld \
    RANLIB=/usr/local/armv7-marvell-linux-gnueabi-hard/bin/arm-marvelllinux-gnueabi-ranlib \
    CFLAGS="-I/usr/local/armv7-marvell-linux-gnueabi-hard/arm-marvelllinux-gnueabi/libc/usr/include -mhard-float -mfpu=
    LDFLAGS="-L/usr/local/armv7-marvell-linux-gnueabi-hard/arm-marvelllinux-gnueabi/libc/lib" \
./configure \
    --host=armle-unknown-linux \
    --target=armle-unknown-linux \
    --build=i686-pc-linux" \
    --prefix=/usr/local
```

For Marvell Armada XP platform in DSM 5.0:

```
env CC=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-gcc \
    LD=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-ld \
    RANLIB=/usr/local/arm-marvell-linux-gnueabi/bin/arm-marvell-linuxgnueabi-ranlib \
    CFLAGS="-I/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/include -mhard-float -mfpu=vfpv3-d16" \
    LDFLAGS="-L/usr/local/arm-marvell-linux-gnueabi/arm-marvell-linuxgnueabi/libc/lib" \
./configure \
    --host=armle-unknown-linux \
    --target=armle-unknown-linux \
    --build=i686-pc-linux" \
    --prefix=/usr/local
```

For Annapurnalabs, Alpine platform in DSM 5.1:

```
env CC=/usr/local/arm-cortex_a15-linux-gnueabi/bin/arm-cortex_a15-linux-gnueabi-ccache-gcc \
    LD=/usr/local/arm-cortex_a15-linux-gnueabi/bin/arm-cortex_a15-linuxgnueabi-ld \
    RANLIB=/usr/local/arm-cortex_a15-linux-gnueabi/bin/arm-cortex_a15-linux-gnueabi-ranlib \
    CFLAGS="-I/usr/local/arm-cortex_a15-linux-gnueabi/arm-cortex_a15-linux-gnueabi/sysroot/usr/include -mfloat-abi=hard
    LDFLAGS="-L/usr/local/arm-cortex_a15-linux-gnueabi/arm-cortex_a15-linux-gnueabi/sysroot/lib" \
./configure \
    --host=arm-cortex_a15-linux-gnueabi \
    --target=arm-cortex_a15-linux-gnueabi \
    --build=i686-pc-linux" \
    --prefix=/usr/local
```

For Mindspeed, Comcerto, C2000 platform in DSM 5.0:

```
env CC=/usr/local/arm-cortexa9-linux-gnueabi/bin/arm-cortexa9-linuxgnueabi-ccache-gcc \
    LD=/usr/local/arm-cortexa9-linux-gnueabi/bin/arm-cortexa9-linuxgnueabi-ld \
    RANLIB=/usr/local/arm-cortexa9-linux-gnueabi/bin/arm-cortexa9-linuxgnueabi-ranlib \
    CFLAGS="-I/usr/local/arm-cortexa9-linux-gnueabi/arm-cortexa9-linuxgnueabi/sysroot/include -mcpu=cortex-a9 -march=ar
    LDFLAGS="-L/usr/local/arm-cortexa9-linux-gnueabi/arm-cortexa9-linuxgnueabi/sysroot/lib" \
./configure \
    --host=armle-unknown-linux \
    --target=armle-unknown-linux \
    --build=i686-pc-linux" \
    --prefix=/usr/local
```

For Intel X86 platform in DSM 5.0:

```
env CC=/usr/local/i686-pc-linux-gnu/bin/i686-pc-linux-gnu-gcc \
    LD=/usr/local/i686-pc-linux-gnu/bin/i686-pc-linux-gnu-ld \
    RANLIB=/usr/local/i686-pc-linux-gnu/bin/i686-pc-linux-gnu-ranlib \
    CFLAGS="-I/usr/local/i686-pc-linux-gnu/i686-pc-linux-gnu/sysroot/usr/include" \
    LDFLAGS="-L/usr/local/i686-pc-linux-gnu/i686-pc-linux-gnu/sysroot/lib" \
./configure \
    --host=i686-pc-linux-gnu \
    --target=i686-pc-linux-gnu \
    --build=i686-pc-linux \
    --prefix=/usr/local
```

For Intel Atom Evansport platform in DSM 5.0:

```
env CC=/usr/local/i686-pc-linux-gnu/bin/i686-pc-linux-gnu-gcc \
    LD=/usr/local/i686-pc-linux-gnu/bin/i686-pc-linux-gnu-ld \
    RANLIB=/usr/local/i686-pc-linux-gnu/bin/i686-pc-linux-gnu-ranlib \
    CFLAGS="-I/usr/local/i686-pc-linux-gnu/i686-pc-linux-gnu/sysroot/usr/include" \
    LDFLAGS="-L/usr/local/i686-pc-linux-gnu/i686-pc-linux-gnu/sysroot/lib" \
./configure \
    --host=i686-pc-linux-gnu \
    --target=i686-pc-linux-gnu \
    --build=i686-pc-linux \
    --prefix=/usr/local
```

Compile Kernel Modules

As mentioned before, you have to create the SynoBuildConf/build, SynoBuildConf/install, and SynoBuildConf/depends before using Package Toolkit.

In this chapter, we will use **platform 6281** as our example.

Preparation:

First, you will need to download the GPL source code of your platform. You can download the source code from this link.

After downloading the Synology GPL kernel source code, move the file to your **toolkit** folder.

SynoBuildConf/depends:

There is nothing special about the **depends** file. The following is the **depends** file for the 6281 platform.

```
[default]
all="6.0"
```

SynoBuildConf/build:

Before compiling the kernel modules, you will need to set up the config file first.

In the kernel source code, there are configuration files for different platforms. The configuration files are listed below:

CPU	Configuration File	Arch	Linux
Marvell 6281 Marvell 6282	synoconfigs/88f6281	ARM	2.6.32
Marvell Armada 370	synoconfigs/armada370	ARM	3.2.40
Marvell Armada 375	synoconfigs/armada375	ARM	3.2.40
Marvell Armada XP	synoconfigs/armadaxp	ARM	3.2.40
Annapurnalabs, Alpine AL212/AL314/AL514	synoconfigs/alpine (DS715, DS1515, DS2015xs) synoconfigs/alpine4k (DS215+, DS416)	ARM	3.2.40
Mindspeed, Comcerto, C2000	synoconfigs/comcerto2k	ARM	3.2.40
Freescale QorIQ (P1022)	synoconfigs/ppcQorIQ	PowerPC	2.6.32
Intel Atom D525, D510, D410, D425	synoconfigs/x86_64	x86	3.2.40
Intel Atom D2700	synoconfigs/cedarview	x86	3.2.40
Intel SoC CE5335	synoconfigs/evansport	x86	3.2.40
Intel Core i3	synoconfigs/bromolow	x86	3.2.40

Please copy the proper configuration file to `.config`, and run `make oldconfig` and `make menuconfig` to select your kernel modules. According to the platforms you would like to compile, you will have to set the proper `ARCH` and `CROSS_COMPILE` into environment variables.

The following is a sample script that will compile the linux kernel modules for the 6281 platform.

```
#!/bin/bash
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.
# SynoBuildConf/build

case ${MakeClean} in
    [Yy][Ee][Ss])
        [ -f Makefile ] && make distclean
        ;;
esac

case ${CleanOnly} in
    [Yy][Ee][Ss])
        return
        ;;
esac

# prepare config files
cp -f synoconfigs/88f6281 .config
make ARCH=${ARCH} CC=${CC} oldconfig

# start compile kernel modules
echo "=====Build Synology Linux kernel 2.6 Modules ====="
make ARCH=${ARCH} CC=${CC} LD="${LD}" ${MAKE_FLAGS} modules

# create table
./scripts/syno_gen_usbmodem_table.sh create-table
```

SynoBuildConf/install

Unlike the previous example, this example will not pack the whole kernel module into one single spk file. Instead, it will install the kernel modules in **/images/modules** under the chroot environment. As a result, the installation script below is slightly different from the previous example.

```
#!/bin/bash
# Copyright (c) 2000-2015 Synology Inc. All rights reserved.
# SynoBuildConf/install

MODULES_DIR="${ImageDir}/modules"

PrepareDirs() {
    [ -d "${MODULES_DIR}" ] || mkdir -p ${MODULES_DIR}
    rm -f ${ImageDir}/modules/*
}

InstallModules() {
    make ARCH=${ARCH} CC=${CC} LD="${LD}" INSTALL_MOD_PATH=$MODULES_DIR modules_install
}

main() {
    PrepareDirs
    InstallModules
}

main "$@"
```

Compile Kernel Modules:

Lastly, run the following commands to compile and install the kernel modules to the destination folder.

```
/toolkit/pkgscripts/PkgCreate.py -p 6281 -c linux-2.6.32
```

After the build process, you can check the result in `/toolkit/ds.6281-6.0/image/modules` .

Verify the Result:

You can copy the kernel module you need for your DSM system and run the following command to install the module.

```
insmod ${module_name}
```

Use the following command to verify that your module has been installed properly.

```
lsmod | grep ${module_name}
```