Gomtuu Build System

User's Guide

June 2019

This guide describes how to setup and use the Gomtuu Build System to build collections of software packages.

STATUS
Draft

This document was prepared using LaTeX 2_ε.

Contents

Preface **ix**

1 Introduction **1**
1.1 Goal . . . 1
1.2 Build Process . . . 1
1.3 Build Program . . . 3

2 Setting up the Build System **5**

3 Defining a New Software Set **9**
3.1 Creating a Software Set . . . 9
3.1.1 Software Set Settings . . . 10
3.1.2 Toolchain Definitions . . . 11
3.2 Populating a Software Set . . . 13
3.2.1 Software Package Settings . . . 13
3.3 Copying a Software Set . . . 16

4 Generating Software Set Contents **17**

5 Creating a Software Set Image **19**

6 Software Set Maintenance **23**
6.1 Retrieving Information . . . 23
6.2 Editing Configurations . . . 23
6.3 Cleaning . . . 24
6.4 Removing Configurations . . . 24

7 Customizing the Build **27**
7.1 Software Set Directory Structure . . . 27
7.2 Build Phase Scripts . . . 30
7.3 Init Phase Scripts . . . 34
7.4 Finalize Phase Scripts . . . 35
7.5 Clean Phase Scripts . . . 35
7.6 Replacing Default Build Phase Scripts . . . 36

A Usage of Executables **37**
A.1 gbs . . . 37
A.2 gbsswim . . . 38

B Default Build Phase Scripts **39**
B.1 Import Script . . . 39
B.2 Patch Script . . . 40
B.3 Configure Script . . . 40
B.4 Build Script . . . 41
B.5 Test Script . . . 41
B.6 Install Script . . . 41
B.7 Deploy Script . . . 41

C Default Clean Phase Scripts **43**
C.1 Clean Script . . . 43

D Build Environment **45**
D.1 Support Scripts . . . 45
D.1.1 Shell Environment . . . 45
D.1.2 File Patching . . . 46
D.1.3 Package Handling . . . 46
D.2 Environment Variables . . . 47

E Examples **51**
E.1 SWSET-1 . . . 51
E.2 SWSET-2 . . . 54

List of Tables

2.1 LOG Setting Values 6

7.1 Build Phase Script Names 30
7.2 Shell Script Exit Codes 30
7.3 Build Script `run-parts` 31
7.4 Install Script `run-parts` 32
7.5 Deploy Script `run-parts` 33
7.6 Init Script Copying sys-root 34
7.7 Finalize Script `99_done.sh` 35
7.8 Clean Phase Script Names 35
7.9 Clean Script `run-parts` 36

D.1 Shell Variables 45
D.2 Shell Functions 45
D.3 File Patching Functions 46
D.4 Package Handling Functions 47
D.5 Environment Variables Build Phase Scripts 47
D.6 Environment Variables Init Phase Scripts 48
D.7 Environment Variables Finalize Phase Scripts 49
D.8 Environment Variables Clean Phase Scripts 49

List of Figures

5.1 Software Set Build Flow 21

7.1 Software Set Directory Layout 29

Preface

This document describes how to setup and use the Gomtuu Build System to create your own Software Sets from a collection of software packages. The installation of the program is not covered by this manual, refer to the `INSTALL` file elsewhere in this distribution for further information on that topic.

Chapter 1

Introduction

1.1 Goal

GBS came into being by the desire to automate the build procedure of a so-called "LinuxFromScratch" system for my Alpha machines. Manually building on older Alpha machines takes a lot of time and duplicating a configuration to another machine is cumbersome. With GBS I've tried to create a build system which solves these issues while still remain simple, lightweight, and preferrably architecture-independent. The starting-point of GBS is that every software package has its own way of building. Many packages however, use the same build method so GBS provides default behaviour. The end-user is free to replace parts of the default behaviour to fit his own needs.

1.2 Build Process

The build process of GBS is essentially the build of one or more software packages for a given target system with some pre- and post-operations added. The end-result is an "installation area" for the target system which can then be copied to the target or packaged into an archive file for further distribution. By itself, GBS doesn't build any software packages, it merely triggers the build system of each individual software package. The package may have a build system of its own — most packages do — or a custom build system which is added by the end-user.

When a number of software packages are grouped together for a specific purpose then this is referred to as a *Software Set*. For example, software packages providing core system functionality could be grouped into a "Base O/S" Software Set. Using Software Sets is however not optional. Every software package you want to build for a given target system must be part of a Software Set.

Before the build of a Software Set begins, GBS allows some preparations to be made during the so-called *init* phase. Typically, this phase creates a directory hierarchy for the target installation (i.e., the "rootfs" of the target system), or copies specific files to the target installation needed for the build. The Init phase consist of 1 step (see below) and is always user-defined (i.e., the build system does not implement default behaviour).

0. Init – initializing software set

The actual build of a package is divided into a number of *build* phases. These phases cover everything from retrieving a package to their final integration on the target installation. There are 7 different build phases, numbered from 0 to 6, which are executed in ascending order when a package is being built. GBS implements default behaviour for all build phases except 6 (Deploy). Although this phase has a specific purpose, the default behaviour is a NOP (no-operation).

0. Import – transfer package sources to build system
1. Patch – apply changes/fixes to package sources
2. Configure – configuring package
3. Build – compiling package
4. Test – validating build
5. Install – transfer package to target installation
6. Deploy – setup package on target installation

For each package GBS builds, it sequentially runs through the above build phases. In case some fatal error occurs, the entire build is terminated prematurely.

After all packages of a Software Set have been built, GBS runs through one last phase, which is the *finalize* phase (see below). As with the Init phase, this phase also consist of 1 step and is also user-defined. The Finalize phase allows some last operations to be performed such as straightening out file permissions and ownership.

0. Finalize – finalizing software set

Contrary to the phases mentioned above, the build system also defines a *clean* phase for removing "build objects" generated during the build. The build system implements default behaviour for this phase.

0. Clean – remove build objects of package

In order to generate code for a target system, GBS uses a toolchain. Toolchains are however not an integral part of the build system, so the end-user has to set this up by himself for the appropriate target system(s). Which toolchain is used for a Software Set is determine at the time of its creation (see Chapter 3).

1.3 Build Program

The main program of the build system is the executable `gbs` which should be located in the system's search paths. Executing `gbs` without any parameters causes a help text to be output. This shows the commands available for creating, building, or cleaning Software Sets. The syntax of `gbs` is:

```
gbs [COMMAND [PARAMETER]...]
```

Almost all commands require a Software Set to operate on, and this is specified as a parameter on the command line. Which parameter, and how many parameters are needed depends on the command. The order in which a command and its parameter(s) are specified on the command line is irrelevant. See Appendix A for the help text output or run `gbs` and specify your first command:

```
gbs help
```

Chapter 2

Setting up the Build System

In order to create Software Sets, GBS needs directory paths where Software Set configurations and "build objects" can be found or stored. In addition, there are some settings which control the behaviour of the build system while it's operating. There are builtin default values which may suffice, but you may want to override them on a system-wide basis or on an individual basis. The system-wide settings are stored in the file

```
<prefix>/etc/gbs.conf
```

where `<prefix>` denotes the installation prefix of GBS. A regular user can put his preferences in his home directory:

```
<home>/.gbs_conf
```

where `<home>` is the home directory of the user. Both of these files may contain definitions for the same settings, and in such a case the user-defined definition will override the system-wide definition, which in turn overrides the builtin default. Note that this only applies to the settings that you actually redefine, so you only need to put the settings you want to change in your `.gbs_conf` file or in the system-wide `gbs.conf` file.

Following is an enumeration of the settings recognized by GBS along with their builtin default value (in parenthesis). The setting identifiers are case-sensitive.

- `ROOTFS_LEVEL` (`/var/lib/gbs/rootfs`)

  Base directory where each Software Set will have its software packages installed, after they have been built. Each Software Set has its own sub-directory in the base directory to keep Software Sets apart.

- `SOURCE_LEVEL` (`/usr/src`)

  Base directory for extracting and building the software packages of a Software Set. Each Software Set has its own sub-directory in the base directory. For each software package of the Software Set, another directory is created within the Software Set directory. Every Software Set has its own copy of package sources.

- `CONFIG_LEVEL` (`/var/lib/gbs/swset`)

  Base directory which contains the configuration of a Software Set. A sub-directory for the Software Set is created in the base directory. The configuration of a Software Set includes a list of software packages contained in the Software Set as well as definitions on how to build each package. Optionally, there may be (static) files or additional instructions present which are needed while the Software Set is being created. The `CONFIG_LEVEL` directory typically holds data that can't be reproduced, contrary to the `ROOTFS_LEVEL` directory and the `SOURCE_LEVEL` directory.

- `PACKAGE_LEVEL` (`ftp://alpha.gomtuu.net/PKG`)

  This setting lists directories and/or URLs where software packages can be found. The directories or URLs are separated by whitespace and are processed (i.e., used) in left-to-right order.

- `GBS_OPT` (*build jobs = #CPUs, log to terminal, purge*)

  The setting is a list of options, each separated by whitespace, and control some behaviour of the build system while its running. The following options are recognized:

  - `JOBS=<n>`

    Determines the maximum number of jobs running in parallel. What these jobs comprise is unspecified. This could be building two software packages in parallel, or compiling multiple source files of the same package in parallel.

  - `LOG=<output>`

    Controls where the output of the build phases is sent to. Only the activity of the build phases can be logged. Other activities, such as cleaning, are usually echo'ed to stdout. Values for `<output>` have the following meaning:

    Table 2.1: LOG Setting Values

no	Mute output
term	Send output to terminal
file	Send output to file corresponding to build phase
cfile	Same as `file` except log file is compressed afterwards

    The values are case-sensitive and default to `term` if omitted.

    If output is sent to file, each build phase is logged to a separate file with the sequence number of the build phase encoded in the filename. The log files of a given software package are stored in a hidden directory in the package's source directory.

  - `PURGE=<yes|no>` (no)

    This option controls whether or not the build system should remove the build

objects of a software package after it has been installed. This operation would occur immediately after the installation of the package and is useful to conserve disk space. Note that administrative files such as log files, are never removed by the `PURGE` option.

- `DEBUG=<yes|no>` (no)

  This is mainly for debugging purposes as it enables or disables echoing the commands of the build system itself as well as the commands of the shell scripts.

Chapter 3

Defining a New Software Set

GBS has three commands for creating a new Software Set. One of them creates a Software Set from scratch while the other two use an existing Software Set as a base model.

3.1 Creating a Software Set

Creating a Software Set from scratch is achieved by using the `create` command of the `gbs` program. This command takes the name of the new Software Set and a base directory of the toolchain to use for building the (future) software packages within the Software Set. The syntax of the `create` command is as follows:

```
create S=<swset_name> {R=<toolchain_dir> | P=<pkg_name>}
```

The name of the Software Set has a limitted range of allowable characters. The characters allowed are: capitals, digits, underscore, and minus sign. The minus sign, however, cannot appear as the first character of the Software Set name.

As an example, here's how I create a Software Set for an AlphaServer 800 system of mine which uses an Alpha EV56 microprocessor.

```
gbs create S=AS800 \
        R=/opt/toolchains/alphaev56-gomtuu-linux-gnu
```

This will create a sub-directory `AS800` in the `CONFIG_LEVEL` directory. In addition, two files will be created in the `AS800` directory: `Defines.mk` and `Toolchain.mk`. The first contains some administrative settings like a description for the Software Set, or how to log the build output. This file may need some manual adjustment (see Section 3.1.1). The `Toolchain.mk` file holds the settings for the toolchain being used and does not need any adjustment, usually. Both of these files use `Makefile` syntax, but it's recommended to limit variable definitions to simple substitutions. Comments can be introduced by a line starting with a '`#`' sign, while the '`\<newline>`' sequence indicates a line-continuation.

3.1.1 Software Set Settings

The administrative settings of a Software Set in the file `Defines.mk`, may be edited by hand when the build system is not operating on the Software Set. Although editing the file while GBS is operating on a Software Set is harmless, modifications will not be picked up until the next time GBS is run since the file is only read once. The following settings are recognized for the `Defines.mk` file.

- `SWSET_ID`

  An ID used to discriminate the Software Set from others wherever needed. For example, the Software Set ID is used in the name of the manifest file of the Software Set.

- `SWSET_VER`

  Version number for this Software Set configuration. The numbering scheme is user-defined.

- `SWSET_DESC`

  Description for this Software Set, typically a one-liner.

- `SWSET_REF`

  Name of a Software Set to be referenced for configuration settings in this Software Set. The name must refer to a non-shadowed Software Set. This option is only used when creating a shadow copy of a Software Set (see also Section 3.3).

- `SWSET_OPT`

  This setting takes the same options as the `GBS_OPT` setting defined in Chapter 2. However, the *options* of the `SWSET_OPT` setting take precedence over those of the `GBS_OPT` setting. So, this setting allows defining Software Set specific options, but mind that only listed options overwrite their counterpart in `GBS_OPT`.

- `EXCLUDE`

  The `EXCLUDE` setting allows (temporary) exclusion of a software package from the build. The setting holds a whitespace-separated list of package names to exclude. Refer to Section 6.1 on how to obtain the package names of a Software Set.

  Excluding packages may cause the build to fail.

- `IMAGE_CMD`

  If you plan on using GBS to generate an image file of your Software Set, then you must specify here the command for your image tool. Any switches your image tool may need can be added as well as a number of format specifiers which GBS will substitute prior to executing the image tool. Format specifiers that are recognized are listed in the table below.

Note that a format specifier such as %V or %L can also be replaced by their respective setting (SWSET_VER and SWSET_ID). It was however chosen to use a uniform method of substituting values in the IMAGE_CMD setting and therefore introduce specifiers such as %V and %L.

- IMAGE_FILE

  Optional pathname of image file. This setting is directly related to the %I format specifier of IMAGE_CMD. The build system substitutes the value of IMAGE_FILE for every %I in the IMAGE_CMD setting. An absolute directory in IMAGE_FILE is left as-is while a relative path is taken to be relative to the current directory.

  It is permissable to omit the filename of the image, meaning you'd only specify a directory path. In this case the build system formats a default filename which is concatenated with the directory path. The directory path must end with a '/' for it to be recognized as a directory else the last element of the directory is taken to be a filename. A default filename for an image has the following form:

  ```
  <swset_id>-<swset_version>
  ```

  If the %I specifier is not employed, the IMAGE_FILE setting is not needed. In such a case the image tool is likely to have an alternate method of naming the image file. Note that the %I specifier can still be used even without specifying an image file. In this case GBS will format a default filename for the image relative to the current directory.

3.1.2 Toolchain Definitions

The Toolchain.mk file defines which toolchain to use and where it can be found. When a Software Set is created, the build system uses a certain scheme where specific parts of the toolchain can be found. GBS assumes that the "target triplet", which identifies the toolchain, is encoded in the base directory of the toolchain. The target triplet follows the convention <cpu-vendor-os> and is the last element in the base directory (see the example earlier in this chapter where a Software Set is created). The target triplet is also assumed to be encoded in the name of certain toolchain executables.

Another important issue is the presence of a "sys-root" directory which a.o. holds libraries and header files for the target installation. The internal layout of the sys-root directory depends on the toolchain. If your toolchain uses a different scheme you obviously need to adjust the Toolchain.mk file. The settings below can be found in the Toolchain.mk file; defaults are in parenthesis.

- GBS_TC_TRIPLET (*last element of GBS_TC_LEVEL*)

  String identifying the toolchain to be used. This is usually in the form of a target triplet.

- `GBS_TC_LEVEL` (*value of "R" parameter*)

  Base directory of toolchain. The value of this setting is set by the "R" parameter of the `gbs` program when a Software Set is created.

- `GBS_TC_BINDIR` (`<GBS_TC_LEVEL>/bin`)

  Directory where user-executables of toolchain are kept.

- `GBS_TC_SYSROOT` (`<GBS_TC_LEVEL>/<GBS_TC_TRIPLET>/sys-root`)

  Directory where the sys-root of the toolchain is located.

- `CC` (`<GBS_TC_TRIPLET>-gcc`)

  Filename of 'C' compiler executable.

- `LD` (`<GBS_TC_TRIPLET>-ld`)

  Filename of linker executable.

- `AR` (`<GBS_TC_TRIPLET>-ar`)

  Filename of archiving executable.

- `AS` (`<GBS_TC_TRIPLET>-as`)

  Filename of assembler executable.

- `CPP` (`<GBS_TC_TRIPLET>-cpp`)

  Filename of preprocessor executable.

- `CXX` (`<GBS_TC_TRIPLET>-g++`)

  Filename of C++ compiler executable.

- `RANLIB` (`<GBS_TC_TRIPLET>-ranlib`)

  Filename of archive index generator.

- `NM` (`<GBS_TC_TRIPLET>-nm`)

  Filename of executable for listing object file symbols.

- `STRIP` (`<GBS_TC_TRIPLET>-strip`)

  Filename of executable for removing symbols from object files.

- `OBJDUMP` (`<GBS_TC_TRIPLET>-objdump`)

  Filename of executable for listing object file information.

3.2 Populating a Software Set

The next task is to populate the Software Set with software packages. This can be done by using the `create` command introduced in the previous section, and specify the "P" parameter to give the name of a software package. The "R" parameter cannot be used in this case, so it's not possible to create a new Software Set and a software package at the same time; this has to be done with two separate `create` commands. For convenience, we cite the command's syntax again:

```
create S=<swset_name> {R=<toolchain_dir> | P=<pkg_name>}
```

Adding a package to a Software Set with `gbs` is pretty straightforward as the following example demonstrates. The example shows two packages being added to the `AS800` Software Set (the `termcap` package and the `bash` package) where one package (`bash`) depends on the other (`termcap`). The dependency aspect of this has to be edited manually in the appropriate configuration file, though.

```
gbs create S=AS800 P=termcap

gbs create S=AS800 P=bash
```

Using a "strategic" name for a software package reduces maintenance since GBS will, by default, use that name to automatically locate the package in the predefined locations (refer to `PACKAGE_LEVEL` in Chapter 2).

3.2.1 Software Package Settings

Every software package added to a Software Set gets its own sub-directory within the Software Set. Package-specific files can be stored there and one file that will always be present is a configuration file for the package. This file is named `Config.mk` which allows fine-grained control over how the package will (or should) be handled. Leaving the file as-is will usually result in a standard build of the package. So, at this point we've created a very minimal Software Set. Starting to build the packages in the Software Set is discussed in the next chapter.

The format of the `Config.mk` file is a simple name/value pair structure. Despite the extension of this file, it is not a `Makefile`.[1] Adding comments or wrapping long lines is supported in `Config.mk` and can be achieved by using the '`#`' sign resp. '`\<newline>`' sequence. There is basically no limit to the length a value can have in `Config.mk`.

Below is an enumeration of the settings used for configuring software packages. The names of the settings in the `Config.mk` file are called qualifiers and they are always in uppercase notation. The format of a setting is as follows:[2]

[1] Prior to GBS version 0.5, `Config.mk` actually was a `Makefile` which explains the extension.

[2] GBS version 0.4 and lower encoded the name of the package in the setting name. Refer to earlier versions of the user manual for the format.

```
<qualifier> = <value>
```

- SRC

  Specifies the origin of the package (local or remote). A local source is basically some directory path, while a remote source can be an ftp site or a remote login. A local source can be one of the following:

  - <path>
  - rcs://<path>
  - sccs://<path>
  - git+file://<host>/<path>

  The following remote sources are supported:

  - ftp://<host>/<path>
  - http://<host>/<path>
  - https://<host>/<path>
  - scp://<host>/<path>
  - sftp://<host>/<path>
  - cvs://<repo-spec>
  - svn://<host>/<path>
  - git://<host>/<path>
  - rcp://<host>/<path>

  Note that secure protocols like https and scp may prompt the user for a password. In such a case the build of the package will stall until a password has been entered.

  This setting is optional and works in conjunction with the setting PACKAGE_LEVEL. If the SRC qualifier is used for a package, the specified source will be searched before the sources under PACKAGE_LEVEL.

- NAM

  Name of package or resource. Usually the name of the package is derived automatically by GBS, but this qualifier allows one to deviate from that behaviour. This may be necessary if there's an upper- or lowercase discrepancy in the filename, or if the build system cannot use wildcards in the filename to locate the package. The NAM qualifier is used to circumvent automatic name selection by allowing the *exact* name of the file or resource to be specified.

  Revisions and labels used by version control systems are also supported by the NAM qualifier. In this case the revision or the label is a suffix of the name enclosed in braces (e.g., name{rev1-2}).

  If the NAM qualifier is used, the VER qualifier is ignored.

- `VER`

  Version of package to use. The latest available version is automatically selected if `VER` is not specified.

- `OPT`

  Set certain build system options to control the build of this package alone. See the `GBS_OPT` setting in Chapter 2 for a list of available options. Options set at package level take precedence over options set at Software Set level and build system level. Only listed options overwrite their counterpart in `GBS_OPT` or `SWSET_OPT`.

- `DEP`

  List of package names (within the Software Set) this package depends on. The package names correspond to the name given when the package configuration was added to the Software Set. It's important to realize that a dependency is not solely determined by another package's header files or libraries, but may also include the presence of, for example, a configuration file on the target installation. This means you have to take all aspects of a possible dependency into account, up to its Deploy phase. However, it's probably not needed to go as far as considering *runtime dependencies* for this setting; limitting yourself to *build-time dependencies* is usually sufficient. Refer to Section 6.1 on how to obtain the package names of a Software Set.

- `CFG`

  This qualifier holds the options used to configure the package. By default, the options are passed to the `./configure` script of the package's build system. The value of `CFG` is passed unmodified to the build phases (refer to Section 1.2) and from here it's evaluated once before the configuration options are passed to the `./configure` script. This allows the user to specify configuration options for the `CFG` qualifier in the exact same manner as one would enter them on the command line. There's one exception you have to take into account here: You *cannot* use single quotes to group options or preserve the literal meaning of words, you must use double quotes and/or backslashes.

- `CPL`

  This qualifier may list options for the compiler to allow a finer control over the generated code. The main purpose is to pass debug or optimization flags to the compiler. If not specified, the build reverts to the default flags of the package. The build system does not evaluate the `CPL` setting.

  Note that this option is currently only passed to the 'C' compiler.

3.3 Copying a Software Set

As mentioned earlier in this chapter, there are two other commands which can create a new Software Set. These are the `clone` command and the `shadow` command, and their syntax is as follows:

```
clone  S=<source_swset_name>
       D=<new_swset_name>
       [R=<toolchain_dir>]

shadow S=<source_swset_name>
       D=<new_swset_name>
       [R=<toolchain_dir>]
```

Both commands take the name of a "source" Software Set (i.e., the original one) and the name of the new Software Set to create. Optionally, a different toolchain can be specified for the new Software Set. If omitted, the toolchain of the "source" Software Set is used.

The difference between the two commands is that the `clone` command creates an independent copy of the "source" Software Set (a snapshot) after which each Software Set essentially evolves in its own way. The `shadow` command on the other hand always follows the configuration of the "source" Software Set except that it uses its own toolchain and administrative settings. The main purpose of the `shadow` command is to provide an easy way to support different processors for the same target system. As a precaution, GBS will not allow you to add (or remove) packages from a shadowed Software Set which, in actuality, would alter the "source" Software Set.

Chapter 4

Generating Software Set Contents

In the previous chapter a Software Set was created with two software packages. Those were merely definitions, no package was retrieved or build. This chapter discusses how the software installation for the target system is created from the Software Set configuration. The build system has two commands for this: A `setup` command and a `build` command. The commands have the following syntax:

```
setup S=<swset_name>

build S=<swset_name> [P=<pkg_name>]
```

The purpose of the `setup` command is to prepare the Software Set for the build operation. This could for example include installing the 'C' library and header files from the toolchain into the target installation. The `setup` command takes care of executing the Init phase of the build, but the actual implementation of that phase is user-defined (see Section 7.3).

The command below initializes the `AS800` Software Set introduced in the previous chapter.

```
gbs setup S=AS800
```

Usually, running the `setup` command is only necessary for the first build of a Software Set to ensure the Software Set is "ready" for the build. This includes the case where the entire target installation has been erased.

The `build` command starts generating the actual contents of the Software Set. The build system runs through the build phases (see Chapter 1) for the software packages within the Software Set. After all packages have been built, the build system executes the Finalize phase which provides the opportunity to wrap-up the target installation. The implementation of this phase is also user-defined, but typically includes fixing access rights on files, stripping debug symbols etc (see Section 7.4). Note that the `build` command can also be used to build an individual package (including its dependencies).

In that case, the Finalize phase is not executed; the Finalize phase is only executed for full builds.

The following two examples build both the entire `AS800` Software Set. Remember that the Software Set holds only two software packages in this example, `termcap` and `bash`. Since `termcap` is a dependency of `bash`, it is automatically build (and installed) when `bash` is build.

```
gbs build S=AS800
```

```
gbs build S=AS800 P=bash
```

As the build of the Software Set progresses, messages about the build system's activities may be output to the terminal or stored in a log file. If a log file is used to collect messages, then each build phase is logged to a separate file where the build phase number is encoded in the filename (Section 1.2 enumerates the build phase numbers). The log files along with various administrative files are stored in a hidden sub-directory of a software package's source directory. For example, the log messages of build phase 0 (Import) of the package `bash` in Software Set `AS800` has the following pathname:

```
<SOURCE_LEVEL>/AS800/bash/.gbs/BUILD_PHASE_0.log
```

A log file may also have the extension `log.gz` if the build has been configured to compress log files.

In case the build fails, the log files may provide clues as to what the cause of the failure might be. After the problem is (supposedly) fixed, the `build` command can be retried. The build system will pick up where it failed the last time and continues the build. Dependencies throughout the Software Set are always taken into account, so GBS may rebuild more after the problem has been fixed.

Chapter 5

Creating a Software Set Image

Once a Software Set has been build, it's possible to store all files of the target installation in a single image file. This image file can then be used to distribute or archive a particular build of the Software Set.

GBS does not enforce how an image file is created, but it does allow you to specify a tool which knows how to create the image. This tool is specified in the `Defines.mk` file of a Software Set configuration and can be executed by using the `image` command of `gbs`. Details on how to specify an image tool for your Software Set can be found in Section 3.1.1.

The `image` command basically only requires the name of the Software Set in order to create an image. The syntax of the command is:

```
image S=<swset_name> [I=<image_file>]
```

The optional parameter `I` specifies the pathname of the image file which consists of a directory path and/or a filename. The rules that apply to the `IMAGE_FILE` setting in the `Defines.mk` file (see Section 3.1.1), also apply to the parameter `I`. However, if the parameter `I` is specified, it takes precedence over the `IMAGE_FILE` setting.

As an example we'll execute the `image` command for the `AS800` Software Set and first specify the image file on the command line. This will create the file `AS800.img` in the current directory. Next, we'll omit the parameter `I` which causes GBS to use the image file specified in the `Defines.mk` file (see below). The name of the image file is not specified in `Defines.mk`, only the directory path (note the trailing '`/`'!). GBS formats a default filename which, in this case, will be `AS800-0.1`. The image file will be located in the sub-directory `build/images/` in the user's home directory.

```
gbs image S=AS800 I=AS800.img
```

```
gbs image S=AS800
```

Snippet from the `Defines.mk` file:

```
SWSET_ID   := AS800
SWSET_VER  := 0.1
...

IMAGE_FILE := $(HOME)/build/images/
IMAGE_CMD  := gbsswim -c -d %R -f %I
```

We'll conclude this chapter and the previous 2 chapters with a schematic depicting the course of events when creating and building a new Software Set (see Figure 5.1). The figure also shows when certain data is created or needed by the build system.

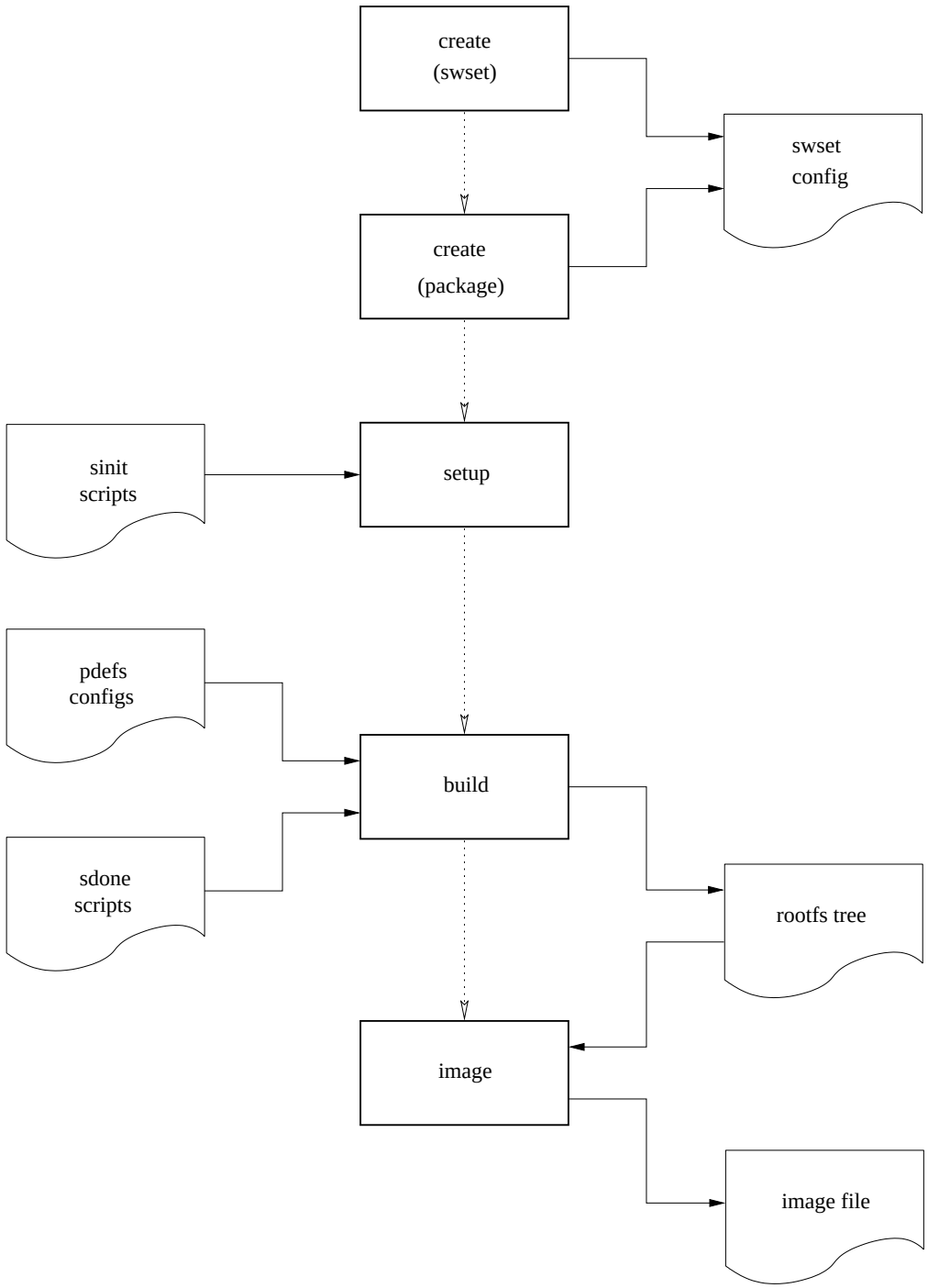

Figure 5.1: Software Set Build Flow

Chapter 6

Software Set Maintenance

GBS provides some additional commands to ease maintenance of the Software Sets. Among others, these commands enable retrieving information about a specific Software Set or software package, or cleaning up files generated at some point of the build.

6.1 Retrieving Information

There's only one command for retrieving information from the build system, the `list` command. It has the following syntax:

```
list [S=<swset_name> [P=<pkg_name>]]
```

Without any parameters the command outputs a concise overview of the available Software Sets. The Software Set names in the output can be used for commands requiring a Software Set name.

The `list` command also accepts a Software Set parameter. In this case, the command outputs details about the Software Set. This includes items like the Software Set's version, the toolchain being used, software packages contained in the Software Set etc. If the software package parameter is added to the command, details about the package as configured in the Software Set, are output. This includes the version of the software package, its source, build options etc. The name of the software package can be retrieved from the Software Set details.

6.2 Editing Configurations

When configuring packages or Software Sets it's sometimes cumbersome to constantly switch working directory or enter long directory paths. As a convenience, GBS provides the `vi` command which lets the build system locate a configuration file and open it in an editor. The command has the following syntax:

```
vi [S=<swset_name> [P=<pkg_name>]]
```

The editor to invoke is determined by the variables `VISUAL` resp. `EDITOR` in the user's shell environment. The variable `VISUAL` takes precendence over the variable `EDITOR`. If neither are defined, the build system uses a builtin default value.

Depending on the parameters specified, the `vi` command opens a configuration file(s) from a different context. This means for example, that when a Software Set and a package name is specified, the configuration file(s) of that specific package are opened. So, you're editing a configuration at package level. Omitting the parameter for the package results in the configuration file(s) of the given Software Set being opened. Finally, when omitting the Software Set parameter, the build system opens up its own configuration file(s).

6.3 Cleaning

Cleaning up files after the build of a Software Set exist in three different commands: `purge`, `clean`, and `realclean`.

```
purge     S=<swset_name> [P=<pkg_name>]

clean     S=<swset_name> [P=<pkg_name>]

realclean S=<swset_name> [P=<pkg_name>]
```

These three commands differ in their severity of removing files. If the software package parameter is specified, the operation only affects the specified package otherwise all packages in the Software Set are affected.

The `purge` command is the least sever kind of removing and will only remove the build objects (typically '`.o`' files and executables). In addition, any sources that have been patched will be restored to their original state. The `clean` command is like the `purge` command but goes one step further: It will also remove administrative files (log files, state files etc). Finally, the `realclean` command will remove the build objects, the administrative files, and the source files. In this case, a software package(s) has to be re-imported before it can be build again.

6.4 Removing Configurations

When the configuration of a software package or an entire Software Set is no longer needed it's possible to permanently erase it. GBS provides the command `remove` to delete a configuration from the `CONFIG_LEVEL` directory. Obviously, this command should be used with caution since configuration data cannot be reproduced by the build system, so once removed it's really gone. The syntax of the `remove` command is:

```
remove S=<swset_name> [P=<pkg_name>]
```

Removing a package from a Software Set also updates the configuration throughout the rest of the Software Set. This is necessary if the package acts as a dependency

of another package. In this case the configuration of the depending package must be updated.

In addition to removing configuration data, the `remove` command also removes the source of a package or a Software Set from the source directory (`SOURCE_LEVEL`). The target installation is never touched by the `remove` command.

It's not possible to remove packages or the entire Software Set from within a shadowed Software Set. The build system will not allow such an action to take place. In this case you'll have to execute the `remove` command from within the originating Software Set.

Chapter 7

Customizing the Build

Thusfar the packages defined in a Software Set were built using default behaviour provided by GBS. However, you may encounter packages which require a different way of building, so you need to be able to divert from GBS its default behaviour and implement your own. What constitutes to default behaviour is described in Appendix B.

This chapter will discuss how to customize the build (or part of the build) of a package. This can be applied to every package in a Software Set, so once you know how to customize one package, you can customize any one of them. In addition to customizing a package, this chapter will also discuss how to implement the Init phase, the Finalize phase, and the Clean phase.

7.1 Software Set Directory Structure

Before adding custom scripts, you should know a thing or two about the directory structure of a Software Set so you can integrate your scripts in the build system.

The configuration directory of a Software Set is given by `<CONFIG_LEVEL>/<swset_name>`, where `CONFIG_LEVEL` is defined in one of the configuration files of GBS, and `swset_name` is the name you gave to your Software Set upon creation (see also Chapter 2 resp. Section 3.1). In the configuration directory of a Software Set there are two files which have already been introduced in Chapter 3, `Defines.mk` and `Toolchain.mk`. Also present are four directories which are named `pdefs`, `sdone`, `sinit`, and `user`. The first directory, `pdefs`, is where the packages of the Software Set are defined. For each package there is a directory under `pdefs` which contains definitions for that particular package. Such a directory is referred to as the package's *definition directory* and is typically named after the package. It holds the `Config.mk` file for the package as well as custom build scripts and patch files. However, only the `Config.mk` file is mandatory, other files are added when necessary.

The directory `sdone` is meant to contain scripts for wrapping up the Software Set (i.e., target installation) once all packages have been built. Directory `sinit` is similar to the directory `sdone` except that the scripts in this directory are meant to be executed after a Software Set is newly created (i.e., you had a clean target installation).

Then there is the directory `user`. This directory is completely user-defined and can therefore contain whatever is additionally required in building the Software Set. This is typically the place to store static files which are copied to the target installation as-is.

In addition to the directories mentioned above, there may be a fifth, optional, directory in the Software Set configuration directory. If present, it's named `sdefault` and contains the default build phase scripts for the Software Set. Refer to Section 7.6 for details.

In Figure 7.1 the general make-up of a Software Set directory is depicted.

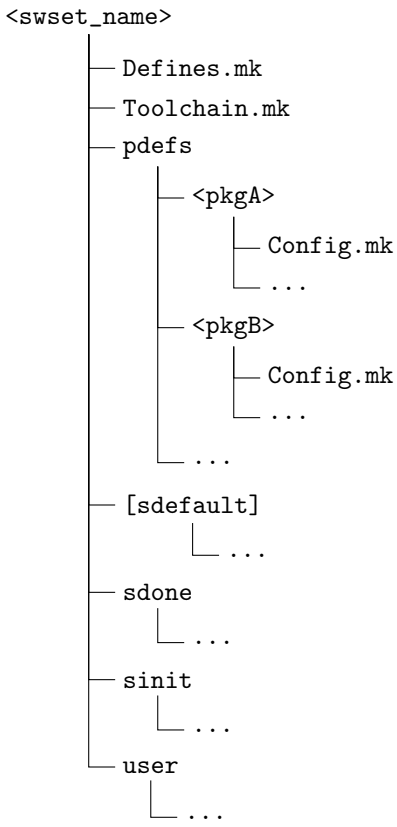

Figure 7.1: Software Set Directory Layout

7.2 Build Phase Scripts

As mentioned in Section 1.2, the build of a package is divided into 7 phases. Behaviour of each build phase is implemented in a separate shell script, each carrying a specific filename. The filenames are fixed irrespective of whether they implement the default behaviour or custom behaviour. The table below lists the exact filenames associated with each build phase.

Table 7.1: Build Phase Script Names

Build Phase	*Script Name*
Import	import.sh
Patch	patch.sh
Configure	configure.sh
Build	build.sh
Test	test.sh
Install	install.sh
Deploy	deploy.sh

By providing a shell script with the proper filename and placing that script in the definition directory of a package, the end-user instructs the build system to use that particular script instead of its default counterpart. Note that this only applies to the concerning package and build phase, other packages are not affected nor are you obliged to provide custom scripts for other build phases. You re-implement what you deem necessary. In addition, the build system also uses the custom script as a *prerequisite* (in `Makefile` slang) in its build process for the package. This means that changes to a custom script will trigger a rebuild of the package from the point of the corresponding build phase.

The actual implementation of a custom script does not matter to the build system, but it usually relates to original purpose of the build phase. The only requirement set by the build system is the exit code of the script. The exit code determines whether or not to continue the build and for your custom script you are responsible for returning an appropriate exit code.

Table 7.2: Shell Script Exit Codes

Exit Code	*Meaning*	*Build Progress*
0	Success	Continue
1	Execution error	Abort
2	Usage error	Abort

In order to assist in writing custom build scripts, GBS provides a number of support functions which, a.o. allow retrieving packages, outputting messages, and managing

patches. Gaining access to these functions is achieved by simply "sourcing" the necessary shell script(s) from the GBS installation in the custom script. Appendix D.1 lists the available functions and a short description.

Another source of assistence useful when writing custom scripts comes in the form of environment variables exported by the build system. Each time the build system executes a build phase script (either custom or default), it has exported a number of variables to the environment which are related to building a package. Some variables are related to the overal build system, just common variables. Other variables are related to building packages or even to the package currently being build. Refer to Appendix D.2, Table D.5 for a list of environment variables related to the build phase.

In Appendix B an overview is given of the operations each of the default build phase scripts performs. This should give an indication of whether or not you need to provide a custom script(s) for a package.

We conclude this section with a simple example of a package which doesn't come with a build system, which means we must provide some custom scripts of our own. It concerns the `run-parts` package from Debian. This package only contains a source file and a man page, so we probably need to provide a custom script where we compile the source file, another script for copying the executable and the man page to the target installation, and finally a script which makes the necessary adjustments to the target system such that `run-parts` can do its duties.

The first script—which compiles and links the executable—should be named `build.sh` and is placed in the definition directory of `run-parts`. If `run-parts` would be part of our `AS800` Software Set which was created in Section 3.1, then the pathname of `build.sh` would be as follows: `<CONFIG_LEVEL>/AS800/pdefs/run-parts/build.sh`. The source code of the shell script is listed below.

Table 7.3: Build Script `run-parts`

```
. $GBS_SCRIPTS_LEVEL/environment.sh                    ❶

env_all || exit 1                                      ❷

for dir in $SWSET_ROOTFS_LIB_LEVEL; do                 ❸
    ldflags="$ldflags -L$dir"
done
for dir in $SWSET_ROOTFS_INC_LEVEL; do
    cppflags="$cppflags -I$dir"
done

jobs_opt=${PKG_JOBS:+-j$PKG_JOBS}                      ❹
${MAKE:-make}   $jobs_opt \                            ❺
                CC=$CC \
                CFLAGS="$PKG_COPTS" \
                LDFLAGS="$ldflags" \
                CPPFLAGS="$cppflags" \
                run-parts
```

❶ Include support functions in build script. The build environment variable `GBS_SCRIPTS_LEVEL` points to the location of the script.

❷ Check environment and create dump of environment. Abort build if environment check failed.

❸ These loops generate a list of library resp. header search paths used for compiling and linking the `run-parts` executable. The variables `SWSET_ROOTFS_LIB_LEVEL` and `SWSET_ROOTFS_INC_LEVEL` hold a list of directories (within the target installation) where libraries resp. header files are located.

❹ Generate a command option for the `make` command which specifies the number of build jobs to start when building the package. Setting the number of build jobs is optional, so the build environment variable `PKG_JOBS` could be unset.

❺ Use builtin rules of `make` to compile and link the executable `run-parts`. Search paths, compiler, and compiler flag are passed to `make` through typical variables like `CC`, `CFLAGS` etc.

The second script to add to `run-parts`'s definition directory should be named `install.sh`. The script basically consists of a copy operation (see below).

Table 7.4: Install Script `run-parts`

```
. $GBS_SCRIPTS_LEVEL/environment.sh                              ❶

dump_env                                                         ❷
env_directories || exit 1

declare -r ubindir="$SWSET_ROOTFS_LEVEL/usr/bin"                 ❸
declare -r man8dir="$SWSET_ROOTFS_LEVEL/usr/share/man/man8"

mkdirhier $ubindir || exit 1
mkdirhier $man8dir || exit 1

install -m755 run-parts   $ubindir || \                          ❹
    fatal "run-parts: failed installing"
install -m444 run-parts.8 $man8dir || \
    fatal "run-parts.8: failed installing"
```

❶ Include support functions in install script. The build environment variable `GBS_SCRIPTS_LEVEL` points to the location of the script.

❷ Check directory-related environment variables and create dump of environment. Abort installation if environment check failed.

❸ Define and create installation directories for executable and man page. Abort installation if directories could not be created. The function `mkdirhier` is a function from the included shell script `environment.sh`.

❹ Copy executable and man page of `run-parts` to the target installation. Should this fail then the function `fatal` outputs the given message and aborts the installation.

The last script to add for `run-parts` is a script for the Deploy phase. The script should be named `deploy.sh` and its function is to integrate the `run-parts` program with the cron facility on the target. The source is shown below.

Table 7.5: Deploy Script `run-parts`

```
. $GBS_SCRIPTS_LEVEL/environment.sh                          ❶

env_all || exit 1                                            ❷

declare -ri cron_gid=16 # group cron                         ❸
declare -r  runpart_dirs="cron.daily cron.weekly \
                          cron.monthly cron.yearly"

for dir in $runpart_dirs; do                                 ❹
    dir="$SWSET_ROOTFS_LEVEL/etc/$dir"
    mkdirhier $dir || exit 1
    chmod 755 $dir
    gbs_chown .$cron_gid $dir || true
done

install -m644 $PKG_CONFIG_LEVEL/user/crontab \               ❺
              $SWSET_ROOTFS_LEVEL/var/adm
```

❶ Include support functions in deploy script. The build environment variable `GBS_SCRIPTS_LEVEL` points to the location of the script.

❷ Check environment and create dump of environment. Abort deployment if environment check failed.

❸ Define two constants used elsewhere in the script.

❹ Create cron job directories managed by `run-parts` on the target installation. The directories are created under `etc` and are owned by the group indicated by number 16 (which is apparently the group number for the cron facility). Here the support function `gbs_chown` is used to change ownership because it performs some common error handling which keeps the deploy script compact.

❺ Copy a crontab for `run-parts` to the target installation. This crontab file is located in the defintition directory of `run-parts` (indicated by the build environment variable `PKG_CONFIG_LEVEL`). (The destination directory may seem strange for a crontab file, but that's not relevant for this example.)

7.3 Init Phase Scripts

The scripts of the Init phase are located in the `sinit` directory and are always user-defined. Their execution is triggered by the `setup` command of GBS, so it's an explicit operation by the end-user. The build system executes the scripts in this directory in alphanumerical order, so if execution order is important then you must enforce the desired order by appropriately naming your scripts. Also note that the scripts must have their execute permissions set; a script without execute permissions set is skipped as part of the Init phase.

Implementing scripts for the Init phase is much the same as implementing build phase scripts. GBS provides support functions and environment variables which can be used in the script. Appendix D.1 lists the available support functions, while Tabel D.6 lists the environment variables exported by the build system. It's also important to take the exit code of the script into account (refer to Tabel 7.2). Failure to do so may cause the build to fail at an unexpected time.

We present one typical init script here. This script shall copy the *sys-root* of the toolchain to the target installation. This is often necessary since packages are building against some core libraries (and headers) provided by the toolchain.

Table 7.6: Init Script Copying sys-root

```
. $GBS_SCRIPTS_LEVEL/environment.sh                          ❶

dump_env                                                     ❷
env_directories || exit 1

readonly BASENAME=‘basename $0 2>/dev/null‘                  ❸

msg "exec $BASENAME"

if test -d "$GBS_TC_SYSROOT"; then                           ❹
    msg "copying sys-root..."
    cp -af $GBS_TC_SYSROOT/* $SWSET_ROOTFS_LEVEL
else
    warn "no sys-root available"
fi
```

❶ Include support functions in init script. The build environment variable `GBS_SCRIPTS_LEVEL` points to the location of the script.

❷ Check directory-related environment variables and create dump of environment. Abort installation if environment check failed.

❸ Define script name constant and echo to stdout.

❹ Check if sys-root directory of toolchain exists which is indicated by the build environment variable `GBS_TC_SYSROOT`. If it does exist, we do a simple copy of the sys-root to the target installation otherwise we emit a warning message.

7.4 Finalize Phase Scripts

Scripts in the `sdone` directory are executed as part of the Finalize phase and are always user-defined. The build system automatically runs through the Finalize phase when all packages of a Software Set have been built, so it's triggered by the `build` command. However, this only applies to a full build of a Software Set, not to an incremental build of individual packages. So, the Finalize phase will not run when you manually build up until the last package in a Software Set.

Here too the scripts in the directory are executed in alphanumerical order, so you need to re-arrange the script names if execution order is important. Also the access permissions of the scripts must at least have the execute bits set; scripts without these permissions are ignored by the build system.

The implementation of scripts for the Finalize phase is along the same lines as the Init phase, except that the build environment variables exported by the build system differ. For the Finalize phase the available variables are listed in Tabel D.6.

As a very simple example we have a script called `99_done.sh` which outputs a message stating that the build of the Software Set was successful. The numeric prefix "99" in the script's filename (hopefully) ensures the script will be executed last in the build.

Table 7.7: Finalize Script `99_done.sh`

```
. $GBS_SCRIPTS_LEVEL/environment.sh

msg ""
msg "!!  SOFTWARE SET COMPLETED  !!"
msg ""
```

7.5 Clean Phase Scripts

Scripts for the clean phase can also be customized and are, just like build phase scripts, package-specific. They are stored alongside those same build phase scripts in the definition directory of the package. The filename of the clean phase script is fixed (see the table below).

Table 7.8: Clean Phase Script Names

Clean Phase	*Script Name*
Clean	clean.sh

The build system will execute the clean phase when the `purge` command is given. Since the `purge` command is also triggered by the `clean` command as well as the `realclean` command, the clean phase is effectively always executed when some cleaning operation is involved.

Implementing a clean phase script works in the same manner as discussed before:

You can use the support functions provided by the build system as well as exported environment variables. In Appendix D.1 a list of support function can be found while the environment variables related to the clean phase are listed in Appendix D.2, Tabel D.8. In order to determine whether the default behaviour of the clean phase is sufficient for your needs, check Appendix C.

The example we present here augments the custom build phase scripts introduced earlier in this chapter (the `run-parts` package). Since this package lacked a build system of its own, we need to provide a script for the clean phase so that `run-parts` is fully integrate in GBS. We therefore create a script named `clean.sh` in the definition directory of `run-parts` with the implementation depicted below.

Table 7.9: Clean Script `run-parts`

```
. $GBS_SCRIPTS_LEVEL/environment.sh                    (1)
. $GBS_SCRIPTS_LEVEL/patch_common.sh

rm -f *.o run-parts                                    (2)

patch_restore                                          (3)
```

(1) Include support functions in clean script. The build environment variable `GBS_SCRIPTS_LEVEL` points to the location of the script.

(2) Remove "build objects" and executable.

(3) If there were files patched they are restored to their original state.

7.6 Replacing Default Build Phase Scripts

Sections 7.2 and 7.5 described how to provide your own script for a certain step in a build or clean phase of a package. In absence of such a script, the build system will use its default counterpart of the script. For each step in a build or clean phase, the build system provides a default script. You may however, want to provide your own default scripts for the build or clean phase such that your software packages can use these scripts as their default. These scripts are stored in the directory `sdefault` of the Software Set configuration directory. This means the scripts will only be used within the context of the given Software Set.

Implementing your own set of default scripts is a matter of populating the `sdefault` directory with *all* the scripts for the build and clean phase, you cannot implement part of the default scripts. The same script names are used when providing per-package build scripts (see Table 7.1 and 7.8).

Note that the `sdefault` directory is not created by GBS, so the end-user has to add this directory to his Software Set when needed.

Appendix A

Usage of Executables

A.1 gbs

```
Usage: gbs [COMMAND [PARAMETER]...]

COMMAND

  create    - add new Software Set or package, <S>{<R>|<P>}
  clone     - duplicate existing Software Set, <S><D>[R]
  shadow    - follow or track existing Software Set, <S><D>[R]
  setup     - prepare Software Set for build, <S>
  build     - build/update Software Set or package, <S>[P]
  image     - generate image file of Software Set, <S>[I]
  purge     - remove build objects of Software Set or package, <S>[P]
  clean     - same as "purge" but also remove administrative files, <S>[P]
  realclean - same as "clean" but also remove sourcecode, <S>[P]
  remove    - delete Software Set or package, <S>[<P>]
  vi        - edit configuration, [<S>[P]]
  list      - show information about Software Sets or packages, [<S>[P]]
  help      - show this help and exit
  version   - show version and exit

PARAMETER

  S  Software Set definition: S=<name>
  D  Software Set definition (duplicate): D=<name>
  P  Package definition: P=<name>
  I  Image definition: I=<pathname>
  R  Toolchain root definition: R=<root>
```

A.2 gbsswim

```
Usage: gbsswim <OPTIONS> <IMAGE>

OPTIONS

  -c        create image of Software Set in directory specified by "-d" option
  -i        install Software Set in image in directory specified by "-d" option
  -t        check image integrity
  -d DIR    use directory DIR (default: .)
  -f        force operation, don't ask questions
  -v        be verbose
  -h        show this help and exit
  -V        show version and exit
```

Appendix B

Default Build Phase Scripts

In the following sections an overview is given about the behaviour of the default build phase scripts which may be executed when a package is being built.

B.1 Import Script

The import script performs the following operations:

- Create dump of environment and verify build environment.
- Determine sources for retrieving package. Supported protocols for a package source include:
  1. `/*` – local path
  2. `ftp://` – ftp URL
  3. `http://`, `https://` – web URL
  4. `scp://`, `sftp://` – secure copy command
  5. `cvs://` – CVS repository
  6. `rcs://` – RCS revision group
  7. `sccs://` – SCCS project
  8. `svn://`, `svn+http://`, `svn+ssh://` – Subversion repository
  9. `git://`, `git+http://`, `git+ssh://`, `git+file://` – GIT repository
  10. `rcp://` – remote copy command
- Scan available sources for requested package. Scan for specific version of package if version was specified, otherwise assume latest version.
- If package is located ensure it's locally accessible.

- A package is assumed to be a `tar` archive (compressed or uncompressed) where the first component of each filename in the archive indicates the "root" directory of the package. This directory is stripped on extraction.
- Extract package in source directory of Software Set unless it's already available.

B.2 Patch Script

This script performs the following operations:

- Create dump of environment and verify build environment.
- Scan directory `patches` in package's definition directory and apply patches. The following conditions apply:
  1. Patch formats recognized:
     - Context diff
     - `ed` script
     - Normal diff
     - Unified context diff
  2. Patch files are handled in alphanumerical order.
  3. Patch files are allowed to reside in a sub-directory under `patches`.
  4. The names of the files to be patched are stripped of one (1) prefix directory (i.e., up to the first slash of the pathname is cut, including the slash itself).
  5. Files to be patched are "secured" in the administrative directory of GBS (allows restoration at a later time).
  6. A successful patch operation is noted in the package's journal.
  7. Assume timestamps of diffs to be UTC.

B.3 Configure Script

The configure script expects to find a GNU `./configure` script in the source directory of the package. The following operations are performed:

- Create dump of environment and verify build environment.
- Add header/library search paths to `CPPFLAGS` resp. `LDFLAGS` variables.
- Set user-defined compiler flags (`CFLAGS` variable).
- Determine host and target system type.
- Execute `./configure` script of package, if present. Host and target system, and user-defined configuration options are passed to the `./configure` script. `CFLAGS`, `CPPFLAGS`, and `LDFLAGS` are exported to the shell environment.

B.4 Build Script

This script expects a `Makefile` in the source directory of the package. Actually, the filename `Makefile`, `makefile`, or `GNUmakefile` is accepted. The script performs the following operations:

- Create dump of environment and verify build environment.
- Determine number of allowed build jobs.
- Run `make` command to start build of package.

B.5 Test Script

The purpose of the test script is currently a bit dubious because of cross-compiling. However, it performs the following operations:

- Create dump of environment and verify build environment.
- Attempt to perform an integrity test of the package. The following targets of the package's `Makefile` are assumed to run the test-suite:
  - `test`
  - `tests`
  - `check`
  - `checks`

  If one of these targets is defined in the `Makefile`, it's executed.

B.6 Install Script

The install script performs the following operations:

- Create dump of environment and verify build environment.
- Run `install` target of package's `Makefile`.
- Use variable `DESTDIR` to pass the root directory of target installation.

B.7 Deploy Script

This script performs the following operations:

- Create dump of environment and verify build environment.

Appendix C

Default Clean Phase Scripts

C.1 Clean Script

The clean script performs the following operations:

- Run `clean` target of package's `Makefile`. The `--keep-going` flag of `make` is employed to clean as much as possible.
- Any source files of the package that have been patched are restored.

Appendix D

Build Environment

D.1 Support Scripts

D.1.1 Shell Environment

Source:

`<GBS_SCRIPTS_LEVEL>/environment.sh`

Table D.1: Shell Variables

Variable	*Meaning*
WHERE	String indicating the current build context.

Table D.2: Shell Functions

Function	*Brief*
env_all	Verify all build phase related variables.
env_directories	Verify whether build phase directory variables are sane.
env_toolchain	Verify whether build phase toolchain variables are sane.
env_build_process	Verify whether build process variables are sane.
dump_env	Produce a dump of the current shell environment.
is_cmd_available	Check whether given command is present on system.

Continued on next page

Table D.2: Shell Functions (*Continued)*

Function	*Brief*
msg	Output common message on standard output channel.
warn	Output warning message on standard error channel.
error	Output message on standard error channel.
fatal	Output message on standard error channel and abort.
trim	Remove leading and trailing whitespace from string.
mkdirhier	Create directory hierarchy.
gbs_chown	Change ownership of file or directory.
fmt_lockfile	Generate filename for a file lock.
gbs_default_patch	Call default "patch" script of GBS.
gbs_default_configure	Call default "configure" script of GBS.
gbs_default_build	Call default "build" script of GBS.
gbs_default_test	Call default "test" script of GBS.
gbs_default_install	Call default "install" script of GBS.
gbs_default_clean	Call default "clean" script of GBS.

D.1.2 File Patching

Source:

`<GBS_SCRIPTS_LEVEL>/patch_common.sh`

Table D.3: File Patching Functions

Function	*Brief*
patch_request	Secure a file for patching.
patch_restore	Restore file(s) that have been patched.
patch_set_mark	Make patch note in journal.
patch_get_mark	Return patch note from journal.

D.1.3 Package Handling

Source:

`<GBS_SCRIPTS_LEVEL>/pkg_retrieval.sh`

`<GBS_SCRIPTS_LEVEL>/pkg_utils.sh`

Table D.4: Package Handling Functions

Function	*Brief*
fetch_local	Retrieve file from local filesystem.
fetch_by_ftp	Retrieve file through FTP session.
fetch_by_http	Retrieve file through HTTP session.
fetch_by_ssh	Retrieve file through SSH channel.
fetch_by_cvs	Retrieve source code from CVS archive.
fetch_by_rcs	Retrieve source code from RCS revision group.
fetch_by_sccs	Retrieve source code from SCCS project.
fetch_by_svn	Retrieve source code from Subversion archive.
fetch_by_git	Retrieve source code from GIT archive.
fetch_by_rcp	Retrieve file through BSD "r"-copy command.
fetch_update_journal	Update fetch activities in journal of current package.
find_latest_version	Given a file pattern, locate the latest version of a file.
extract_version	Filter version number from filename.
extract_revision	Filter revision from filename.

D.2 Environment Variables

Table D.5: Environment Variables Build Phase Scripts

Variable	*Meaning*
GBS_LEVEL	Base directory of GBS files.
GBS_SCRIPTS_LEVEL	Base directory of GBS support scripts.
GBS_TMPDIR	Directory for temporary files.
GBS_SWSET	Name of current Software Set.
GBS_OPT	Build system options.
MAKE	`make` executable.
GBS_PACKAGE_LEVEL	Base directory of software depot.

Continued on next page

Table D.5: Environment Variables Build Phase Scripts (*Continued)*

Variable	*Meaning*
GBS_THIS_MACHINE	Architecture-vendor-OS string identifying this host.
GBS_TC_TRIPLET	Architecture-vendor-OS string identifying toolchain.
GBS_PACKAGE_SOURCE_LIST	List of local/remote package sources.
SWSET_CONFIG_LEVEL	Base directory of Software Set's configuration.
SWSET_SOURCE_LEVEL	Base directory of Software Set's sources.
SWSET_ROOTFS_LEVEL	Base directory of Software Set's rootfs.
SWSET_ROOTFS_LIB_LEVEL	List of "lib" dirs within Software Set's rootfs.
SWSET_ROOTFS_INC_LEVEL	List of "include" dirs within Software Set's rootfs.
SWSET_USER_LEVEL	Base directory of user-provided files.
SWSET_PACKAGE	Name of current package.
SWSET_PACKAGE_VER	Version of current package.
SWSET_PHASE	Current build phase.
PKG_CONFIG_LEVEL	Config directory of current package.
PKG_SOURCE_LEVEL	Source directory of current package.
PKG_JOURNAL	Summarized log of build.
PKG_JOBS	Max. number of jobs when building package.
PKG_CONFIGURE	Package configuration options.
PKG_COPTS	Package compilation options.

Table D.6: Environment Variables Init Phase Scripts

Variable	*Meaning*
GBS_LEVEL	Base directory of GBS files.
GBS_SCRIPTS_LEVEL	Base directory of GBS support scripts.
GBS_TMPDIR	Directory for temporary files.
GBS_SWSET	Name of current Software Set.
GBS_OPT	Build system options.
MAKE	`make` executable.
GBS_TC_TRIPLET	Architecture-vendor-OS string identifying toolchain.

Continued on next page

Table D.6: Environment Variables Init Phase Scripts (*Continued)*

Variable	*Meaning*
GBS_TC_LEVEL	Base directory of toolchain to use for Software Set.
GBS_TC_BINDIR	Base directory of toolchain's executables.
GBS_TC_SYSROOT	Base directory of toolchain's sys-root.
SWSET_CONFIG_LEVEL	Base directory of Software Set's configuration.
SWSET_SOURCE_LEVEL	Base directory of Software Set's sources.
SWSET_ROOTFS_LEVEL	Base directory of Software Set's rootfs.
SWSET_USER_LEVEL	Base directory of user-provided files.
SWSET_PHASE	Current init phase.

Table D.7: Environment Variables Finalize Phase Scripts

Variable	*Meaning*
GBS_LEVEL	Base directory of GBS files.
GBS_SCRIPTS_LEVEL	Base directory of GBS support scripts.
GBS_TMPDIR	Directory for temporary files.
GBS_SWSET	Name of current Software Set.
GBS_OPT	Build system options.
MAKE	`make` executable.
SWSET_CONFIG_LEVEL	Base directory of Software Set's configuration.
SWSET_SOURCE_LEVEL	Base directory of Software Set's sources.
SWSET_ROOTFS_LEVEL	Base directory of Software Set's rootfs.
SWSET_USER_LEVEL	Base directory of user-provided files.
SWSET_PHASE	Current finalize phase.

Table D.8: Environment Variables Clean Phase Scripts

Variable	*Meaning*
GBS_LEVEL	Base directory of GBS files.
GBS_SCRIPTS_LEVEL	Base directory of GBS support scripts.

Continued on next page

Table D.8: Environment Variables Clean Phase Scripts (*Continued)*

Variable	*Meaning*
GBS_TMPDIR	Directory for temporary files.
GBS_SWSET	Name of current Software Set.
GBS_OPT	Build system options.
MAKE	`make` executable.
SWSET_CONFIG_LEVEL	Base directory of Software Set's configuration.
SWSET_SOURCE_LEVEL	Base directory of Software Set's sources.
SWSET_PACKAGE	Name of current package.
SWSET_PACKAGE_VER	Version of current package.
SWSET_PHASE	Current finalize phase.
PKG_CONFIG_LEVEL	Config directory of current package.
PKG_SOURCE_LEVEL	Source directory of current package.
PKG_JOURNAL	Summarized log of build.

Appendix E

Examples

This chapter presents some examples on how to use GBS and demonstrates certain capabilities of the build system. The examples show which commands need to be executed from the command line (indicated by '$' prompt) and which files need to be edited. Note that you need to edit the files such they'll match with the files in the example. The examples are also included in the source distribution of GBS, in the directory `examples`.

For each example the following `.gbs_conf` file was used which is located in the user's home directory.

```
$ cat ~/.gbs_conf
DEBUG          = no

ROOTFS_LEVEL   = ${HOME}/tmp/gbs/rootfs
SOURCE_LEVEL   = ${HOME}/tmp/gbs/src
CONFIG_LEVEL   = ${HOME}/tmp/gbs/cf
PACKAGE_LEVEL = ${HOME}/PKG
```

As can be seen, all build system activity will occur from within the user's (private) `tmp` directory. The `PKG` directory is where most of the tarballs of the software packages are kept.

E.1 SWSET-1

Creating Software Set:

```
$ gbs create S=SWSET-1 R=/opt/toolchains/alphaev67-gomtuu-linux-gnu
$
$ cat ~/tmp/gbs/cf/SWSET-1/Defines.mk
SWSET_ID        := SWSET-1
SWSET_VER       := 0.1
SWSET_DESC      := Simple Software Set test
SWSET_OPT       := LOG=cfile PURGE=no
```

```
EXCLUDE         := john

IMAGE_CMD       := gbsswim -c -d %R -f %I
$
$ cat ~/tmp/gbs/cf/SWSET-1/Toolchain.mk
# Toolchain.mk -- toolchain definitions
#
GBS_TC_TRIPLET  := alphaev67-gomtuu-linux-gnu
GBS_TC_LEVEL    := /opt/toolchains/alphaev67-gomtuu-linux-gnu
GBS_TC_BINDIR   := $(GBS_TC_LEVEL)/bin
GBS_TC_SYSROOT  := $(GBS_TC_LEVEL)/$(GBS_TC_TRIPLET)/sys-root
CC              := $(GBS_TC_TRIPLET)-gcc
LD              := $(GBS_TC_TRIPLET)-ld
AR              := $(GBS_TC_TRIPLET)-ar
AS              := $(GBS_TC_TRIPLET)-as
CPP             := $(GBS_TC_TRIPLET)-cpp
CXX             := $(GBS_TC_TRIPLET)-g++
RANLIB          := $(GBS_TC_TRIPLET)-ranlib
NM              := $(GBS_TC_TRIPLET)-nm
STRIP           := $(GBS_TC_TRIPLET)-strip
OBJDUMP         := $(GBS_TC_TRIPLET)-objdump
```

Populating Software Set:

```
$ gbs create S=SWSET-1 P=bash
$ gbs create S=SWSET-1 P=gomtuu-rc
$ gbs create S=SWSET-1 P=john
$ gbs create S=SWSET-1 P=ntp
$ gbs create S=SWSET-1 P=readline
$ gbs create S=SWSET-1 P=termcap
$
$ cat ~/tmp/gbs/cf/SWSET-1/pdefs/bash/Config.mk
NAM        =
SRC        =
VER        =
OPT        =
DEP        = readline termcap
CFG        = --prefix=/usr --exec-prefix= \
             --infodir=\${prefix}/share/info \
             --mandir=\${prefix}/share/man --disable-multibyte \
             --disable-net-redirections --disable-restricted \
             --enable-separate-helpfiles --disable-mem-scramble \
             --disable-profiling --disable-nls \
             --with-installed-readline=$SWSET_ROOTFS_LEVEL/usr
CPL        =
$
$ cat ~/tmp/gbs/cf/SWSET-1/pdefs/gomtuu-rc/Config.mk
NAM   = gomtuu-rc{11}
SRC   = svn+http://alpha.gomtuu.net/repos/gomtuu-rc/trunk
VER   =
OPT   =
DEP   =
CFG   =
CPL   =
```

```
$
$ cat ~/tmp/gbs/cf/SWSET-1/pdefs/john/Config.mk
NAM        =
SRC        = ftp://alpha.gomtuu.net/pub/software/util
VER        =
OPT        =
DEP        =
CFG        = --prefix=/usr
CPL        =
$
$ cat ~/tmp/gbs/cf/SWSET-1/pdefs/ntp/Config.mk
NAM = ntp-4.2.6p5.tar.gz
SRC = http://archive.ntp.org/ntp4/ntp-4.2
VER =
OPT =
DEP =
CFG = --prefix=/usr --sysconfdir=/etc --disable-debugging \
      --disable-all-clocks --enable-local-clock --disable-ipv6
CPL =
$
$ cat ~/tmp/gbs/cf/SWSET-1/pdefs/readline/Config.mk
NAM    =
SRC    =
VER    =
OPT    =
DEP    =
CFG    = --prefix=/usr
CPL    =
$
$ cat ~/tmp/gbs/cf/SWSET-1/pdefs/termcap/Config.mk
NAM     =
SRC     =
VER     =
OPT     =
DEP     =
CFG     = --prefix=/usr
CPL     =
$
$ cat ~/tmp/gbs/cf/SWSET-1/pdefs/termcap/install.sh
#!/bin/bash

${MAKE:-make} prefix=$SWSET_ROOTFS_LEVEL/usr install

# must copy termcap file manually to rootfs:
install -m644 termcap.src $SWSET_ROOTFS_LEVEL/etc/termcap
```

Building Software Set:

```
$ gbs setup S=SWSET-1
$ gbs build S=SWSET-1
```

Creating Software Set Image:

```
$ gbs image S=SWSET-1 I=SWSET-1.img
$
```

```
$ ls -l SWSET-1.img
-rw-r--r-- 1 arthur arthur 50606592 Oct 10 17:49 SWSET-1.img
```

Cleaning Software Set:

```
$ gbs clean S=SWSET-1
```

E.2 SWSET-2

Prepare RCS Revision Group:

```
$ mkdir /tmp/rcs_test
$ mkdir /tmp/rcs_test/RCS
$ cat /tmp/rcs_test/hellobuilders.c
#include <stdio.h>
int main(int argc, char **argv)
{
        printf("Hello Builders!\n");
        return 0;
}
$ rcs ci -t-. -i hellobuilders.c
```

Creating Software Set:

```
$ gbs create S=SWSET-2 R=/opt/toolchains/alphaev56-gomtuu-linux-gnu
$
$ cat ~/tmp/gbs/cf/SWSET-2/Defines.mk
SWSET_ID        := SWSET-2
SWSET_VER       := 0.1
SWSET_DESC      := Software Set test with repos
SWSET_OPT       := LOG=cfile PURGE=no

EXCLUDE         :=
$
$ cat ~/tmp/gbs/cf/SWSET-2/Toolchain.mk
# Toolchain.mk -- toolchain definitions
#
GBS_TC_TRIPLET  := alphaev56-gomtuu-linux-gnu
GBS_TC_LEVEL    := /opt/toolchains/alphaev56-gomtuu-linux-gnu
GBS_TC_BINDIR   := $(GBS_TC_LEVEL)/bin
GBS_TC_SYSROOT  := $(GBS_TC_LEVEL)/$(GBS_TC_TRIPLET)/sys-root
CC              := $(GBS_TC_TRIPLET)-gcc
LD              := $(GBS_TC_TRIPLET)-ld
AR              := $(GBS_TC_TRIPLET)-ar
AS              := $(GBS_TC_TRIPLET)-as
CPP             := $(GBS_TC_TRIPLET)-cpp
CXX             := $(GBS_TC_TRIPLET)-g++
RANLIB          := $(GBS_TC_TRIPLET)-ranlib
NM              := $(GBS_TC_TRIPLET)-nm
STRIP           := $(GBS_TC_TRIPLET)-strip
OBJDUMP         := $(GBS_TC_TRIPLET)-objdump
```

Populating Software Set:

```
$ gbs create S=SWSET-2 P=alpha-utils
$ gbs create S=SWSET-2 P=git
$ gbs create S=SWSET-2 P=cvs
$ gbs create S=SWSET-2 P=rcs_test
$ gbs create S=SWSET-2 P=zlib
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/alpha-utils/Config.mk
NAM     = alpha-utils{8}
SRC     = svn+http://unimatrix.gomtuu.net/repos/alpha-utils/trunk
VER     =
OPT     =
DEP     =
CFG     = --prefix=/usr ac_cv_func_malloc_0_nonnull=yes
CPL     = -g -Os
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/git/Config.mk
NAM     = git{v2.2.0}
SRC     = git://github.com/git/git.git
VER     =
OPT     =
DEP     = zlib
CFG     = --prefix=/usr --without-expat --with-zlib=$SWSET_ROOTFS_LEVEL/usr
CPL     =
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/git/configure.sh
#!/bin/bash
. $GBS_SCRIPTS_LEVEL/environment.sh
. $GBS_SCRIPTS_LEVEL/patch_common.sh

ar=$GBS_TC_TRIPLET-ar
cc=$GBS_TC_TRIPLET-gcc
ranlib=$GBS_TC_TRIPLET-ranlib

make configure
AR=$ar RANLIB=$ranlib CC=$cc ./configure $PKG_CONFIGURE
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/cvs/Config.mk
NAM     = ccvs
SRC     = cvs://:pserver:anonymous@cvs.sv.nongnu.org:/sources/cvs
VER     =
OPT     =
DEP     =
CFG     = --disable-server --enable-encryption cvs_cv_func_printf_ptr=yes
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/rcs_test/Config.mk
NAM     = hellobuilders.c{1.1}
SRC     = rcs:///tmp/rcs_test
VER     =
OPT     =
DEP     =
CFG     =
CPL     =
$
```

```
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/rcs_test/build.sh
#!/bin/sh
make hellobuilders
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/rcs_test/install.sh
#!/bin/sh
install -m755 $PKG_SOURCE_LEVEL/hellobuilders $SWSET_ROOTFS_LEVEL/usr/bin
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/rcs_test/clean.sh
#!/bin/sh
rm -f $PKG_SOURCE_LEVEL/hellobuilders
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/zlib/Config.mk
VER     =
OPT     =
DEP     =
CFG     = --prefix=/usr --shared
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/zlib/configure.sh
#!/bin/bash
. $GBS_SCRIPTS_LEVEL/environment.sh
. $GBS_SCRIPTS_LEVEL/patch_common.sh

for d in $SWSET_ROOTFS_LIB_LEVEL
do
        ldflags="$ldflags -L$d"
done

for d in $SWSET_ROOTFS_INC_LEVEL
do
        cppflags="$cppflags -I$d"
done

ar=$GBS_TC_TRIPLET-ar
cc=$GBS_TC_TRIPLET-gcc
ranlib=$GBS_TC_TRIPLET-ranlib

AR="$ar rc" RANLIB=$ranlib CC=$cc ./configure $PKG_CONFIGURE

if patch_request Makefile; then
        ed -s Makefile << _EOF
,s@^[[:blank:]]*LDFLAGS[[:blank:]]*=[[:blank:]]*.*@& $ldflags@
/[[:blank:]]*CPP[[:blank:]]*=[[:blank:]]*.*/
.a
CPPFLAGS=$cppflags
.
,s@^[[:blank:]]*LIBS[[:blank:]]*=[[:blank:]]*.*@& libz.a@
wq
_EOF
fi
$
$ cat ~/tmp/gbs/cf/SWSET-2/pdefs/zlib/install.sh
#!/bin/sh
```

```
make prefix=$SWSET_ROOTFS_LEVEL/usr install
$
$ cat ~/tmp/gbs/cf/SWSET-2/sinit/10_mkdirhier.sh
#!/bin/bash
. $GBS_SCRIPTS_LEVEL/environment.sh

BASENAME=‘basename $0 2>/dev/null‘

R=$SWSET_ROOTFS_LEVEL

echo "$WHERE exec $BASENAME"

if ! test -d "$R"; then
        echo "$WHERE: rootfs directory not found: $R"
        exit 1
fi

echo "$WHERE  checking /usr tree..."
[ -d $R/usr         ] || mkdir $R/usr
[ -d $R/usr/bin     ] || mkdir $R/usr/bin
[ -d $R/usr/include ] || mkdir $R/usr/include
[ -d $R/usr/lib     ] || mkdir $R/usr/lib
[ -d $R/usr/libexec ] || mkdir $R/usr/libexec
[ -d $R/usr/sbin    ] || mkdir $R/usr/sbin
[ -d $R/usr/share   ] || mkdir $R/usr/share

echo "$WHERE  checking /var tree..."
[ -d $R/var         ] || mkdir $R/var
[ -d $R/var/adm     ] || mkdir $R/var/adm
$
$ cat ~/tmp/gbs/cf/SWSET-2/sdone/99_done.sh
#!/bin/bash
. $GBS_SCRIPTS_LEVEL/environment.sh

echo "$WHERE Software Set completed"
```

Building Software Set:

```
$ gbs setup S=SWSET-2
$ gbs build S=SWSET-2
```

Checking Build Results:

```
$ ls -lR ${HOME}/tmp/gbs/rootfs/SWSET-2
<list of installed files>
```

Cleaning Software Set:

```
$ gbs clean S=SWSET-2
```