ProviewR

OPEN SOURCE PROCESS CONTROL

Developer's Guide

2011-01-20

Table of Contents

About this manual

Proview Developer's Guide describes the structure of the Proview source code and how to build the source.

The intended audience for this manual are persons that wants to build Proview from sources, on a specific platform, or to make changes and additions to the functionality.

Introduction

The structure of the Proveiw source code three is made to be able to build on different hardware and on different operating systems. The original design was made for VAXELN on VAX, OpenVMS on VAX and Alpha, and LynxOS on x86 and PowerPC. All these platforms are now cut out and replaced by Linux on x86 and x86_64.

The Proview source code is quite extensive, it contains about 800 000 lines, the most part in c and c++, but also some java. In addition there are a number of other types of files for documentation, help texts, class and object descriptions, object graphs, shellscripts etc.

To build Proview, implies that you from the source code tree creates build tree, where the result of the building is placed. Parts of the build tree is then included in installation packages for development, process, operator and storage stations.

The building is performed by a script, pwre, that calls a set of make files with general rules of how to treat different types of files.

This guide is about how the source code is constructed, and how you use pwre to build it.

Source code

Fetch the source code

The source code is available on SourceForge as a tarball, and from a git repository.

Tarball

Download the tarball, for example pwrsrc_4.7.0-1.tar.gz. Unpack it with the command

tar -xzvf pwrsrc_4.7.0-1.tar.gz

Git repository

To download, or clone, a git repository, you first have to install git (git-core). Download the code with the 'git clone' command, see the Proview homepage for more information (Development, Git Repository).

Source code tree

The source code tree consists of at most 6 levels. The environment variable $pwre_croot points at the root, and the different levels are

`$pwre_croot/module/type/component/flavor/os/hw`

One example is

`$pwre_croot/profibus/lib/rt/src/os_linux/hw_x86`

from which we can read that the module is `profibus`, i.e. a field bus. The type is `lib` (library), that will create an archive. The component is `rt`, which will compose the archive name to libpwr_rt.a that contains modules for runtime. On the directory `src` is placed source code for the archive that is common for all operative systems and hardware. On the directory `os_linux` is placed source code that is specific for the operating system Linux. On the directory `hw_x86` is place code that is specific for the hardware x86 on Linux.

Modules

The source code tree is divided in modules. These are located as directories beneath the root directory. The modules consist of three kernel modules, rt, xtt and wb, that are essential for building a runnable system. They are also mutually dependent on each other.

Kernel modules

Module	*Description*
rt	rt signifies RunTime, and are for historical reasons located under the directory *src*. It

	contains all the basic functions in the runtime environment, e.g. the realtime database, communication programs (qcom, nethandler, subscriptions), plc, event handling, backup, system and base classes. In rt is also located most of the documentation, and tools for building Proview.
xtt	Xtt is located under the directory xtt, and contains code for the graphical interface fo the operator environment, and graphical components for the plc editor and drawing of process graphs.
wb	Wb, WorkBench, contains code for the development environment, i.e. the development database, the configurator, the plc editor, the class editor etc.

Other modules

Module	*Description*
Remote	Code for communication over a number of different protocols: 3964R, Modbus, MQ, RK512, TCPIP, UDPIP, ALCM and serial.
Nmps	A simple material handling system.
Profibus	I/O handling with the fieldbusses Profibus and Profinet.
opc	Communication with other process control equipment over the OPC XML/DA protocol.
java	Java interface to the realtime database, process graphs and web interface etc.
otherio	I/O systems that doesn't have it's own module, e.g. Modbus/TCP, Motion Control USB I/O.
bcomp	BaseComponent. A set of component and aggregate classes for different types of valves, pumps, fans, motors, frequency converters, contactors, circuit breakers etc.
Othermanu	Component classes for manufacturers that doesn't have a specific module.
Abb**,** siemens, ssabox, inor, klocknermoeller, telemecanique	Modules that contains classes for components and I/O units for a specific manufacturer.

Typer

The level below the module is called type, and denotes what type of component that is generated, for exampel if it is an archive, and executable, a classvolume. The type can be exe, lib, wbl, msg,

exp, mmi, doc or tools.

Type	*Description*
exe	Exe is for executable and each component below the exe directory generates an exe-file, i.e. an executable program that is placed in $pwr_exe in the build tree.
lib	Lib is for library, and each component below the lib directory, generates an archive with the name lib_pwr'component'.a, that is placed in $pwr_lib in the build tree.
wbl	Wbl is for workbench loadfiles, that is files for description of object, usually class definition objects describing classes. The objects in a component below wbl constitutes a volume, and each component generates a dbs-file on $pwr_load.
msg	Msg is for message and contains files for status codes (.msg). The division in components corresponds to the archives.
exp	Exp is for export, and contains source code that doesn't fit elsewhere.
mmi	Mmi is equivalent to hmi, and contains files concerning the user interface, e.g. language dependent files and various picture files, object graphs and subgraphs.
jpwr	A typ in the java module, where each component generates a java archive, named pwr_'component'.jar, placed on $pwr_lib.
doc	Documentation, help texts etc.
tools	A type in the rt module that contains various files for building and maintenance of the source code.

Components

The component level normally constitutes a specific unit, that is generated by the code in the directories below the component level. Both source code files and build files are found there. The component level can also be used as an hierarchy to collect files of similar nature.

Exe

For components of type exe, an executable program with a name that equals the component name, is created. For example `$pwr_croot/src/exe/rt_ini` contains code and build files for the program `$pwr_exe/rt_ini`. Below the komponent directory, there are a src directory containing a number of c-files, rt_ini.c, ini.c, ini_rc.c and ini_loader.c, and some includefiles. rt_ini.c, that is the file with the same name as the component name, contains the main function. The other c-files are compiled and linked with the program.

Under `rt_ini/src/os_linux` the file `link_rule.mk` is located, that contains the link command for the program on linux.

Under `rt_ini/src/os_linux/hw_x86` the file `makefile` is located, that includes generic makefiles used to build the program.

Lib

Lib components contains c and c++ files that are compiled and inserted into an archive. If we look closer to the archive $pwre_croot/src/lib/rt, that contatins functions for Proview runtime, the main

part of the code is located in rt/src. The c-files are compiled and inserted into the archive $pwr_lib/libpwr_rt.a, and the include files are copied to $pwr_inc. Some source files, that are specific for the operating system, are located one level lower in the tree, in the directory rt/src/os_linux. On rt/src/os_linux/hw_x86 a makefile is found, that includes generic makefiles to create the archive, compile the c-files and copy the h-files to $pwr_inc.

Wbl

A wbl-component contains wb_load-files with object descriptions. These description generates a volume containing object. Usually the generated volume is a class volume, but there are also some examples of other types of volumes, a SharedVolume (rt) and a WorkbenchVolume (wb). From the wb_load-files a dbs-file is generated on $pwr_load, and an h-file containing c structures for the klasses, and an hpp-file containg c++ classes. Furthermore are help files and postscript-files for documentation in Object Reference Manual generated.

If we look closer at $pwre_croot/wbl/pwrb, that contains the Proview baseclasses in the classvolume pwrb. In the pwrb/src catalog, one wb_load file for each class in the volume are located, for example pwrb_c_and.wb_load for the class And. There are also wb_load files for type definitions, e.g. pwrb_td_yesnoenum.wb_load that is an enumeration type for yes/no. In pwrb/src/os_linux/hw_x86 there is a makefile that generates the dbs-file $pwr_load/pwrb.dbs and athe include-files $pwr_inc/pwr_baseclasses.h and $pwr_inc/pwr_baseclasses.hpp. Also the help-files $pwr_exe/en_us/pwrb_xtthelp.dat and $pwr_exe/sv_se/pwrb_xtthelp.dat in english and swedish are generated. In $pwr_doc/en_us/orm and $pwr_doc/sv_se/orm a set of html-files are placed for the Object Reference Manual, and on $pwr_doc a postscript version of Object Reference Manual.

Another model of volume component is $pwre_croot/bcomp/wbl/bcomp. It contains only one wb_load-file, basecomponent.wb_load, containing all types and classes in the volume Basecomponent. The volume is edited by starting the class editor with the command

```
> wblstart.sh basecomponent
```

Otherwise the build is performed similar to the pwrb volume.

Msg

A msg component consist of a number of msg-files that contains status codes. A status code constitutes of an integer value that can be translated to a single line text. The integer values are defined by a #define state in an include file and can be used in the c code, and the texts are compiled to object modules that is linked to the program, which makes it possible to translate the status code to the text. For example GDH__FILE can be translated to ”No such file”. The status codes have five severity levels, success, info, warning, error and fatal, that are associated with the colors green, green, yellow, red and flashing red. The status codes are used, among other things to indicate status of server processes and applications i Proview. It is also used for return status in c-functions.

If we take a look at the message component $pwre_sroot/msg/rt, it contains msg-files used in the lib component rt. For the message file rt/src/rt_gdh_msg.msg, the include-file $pwr_inc/rt_gdh_msg.h is generated. Furthermore the file $pwr_obj/pwr_msg_rt.o is generated, containing the texts for all the message-files in the rt component. When a program links with this o-file, it is able to translate the status codes to texts.

Exp

Exp contains various files that doesn't fit in any other component. Under exp/inc include-files are located that is copied to $pwr_inc, and under exp/com command and script-files are located, that are copied to $pwr_exe etc.

Mmi

Mmi contains files for the user interface, for example pwg and pwsg files for object graphs and subgraphs. Also uil-files, description files for motif components, are located here. Some language specific files are also found here, see the translation chapter below.

When a mmi-component is buildt, object graphs and subgraphs are copied to $pwr_exe. When building motif, the uil-files are compiled to uid-files by a special uil compiler.

Jpwr

A jpwr-component contains a number of java files that are compiled and inserted into a java archive.

If we take a closer look at the component $pwre_croot/java/jpwr/rt, all the java-files are located in rt/src. While java is independent of platform there is no need to place any java-files on the platform directories below rt/src. When building the component, the java-files are compiled to class-files that is placed in the java archive $pwr_lib/pwr_rt.jar.

jpwr/rt is a java interface to Proview runtime, to communicate over qcom, fetch data from the realtime database, handle events and alarms etc. To call the c-functions for these modules, java native is used. The c-code for the native classes is located in java/exe/jpwr_rt_gdh that generates the so-file $pwr_exe/pwr_rt_gdh.so.

Parallel to the jpwr/rt component, there is a jpwr/rt_client component, where one class, Gdh, that attaches the realtime database, is exchanged. This archive is used instead of jpwr/rt by the web interface and fetches data from the realtime database via a socket and a server process.

Doc

Below doc there is a number of directories to generate Proview documentation. doc/man contains manuals, where for example English versions are placed in doc/man/en_us and Swedish on doc/man/sv_se. Common picture files are placed in doc/man/src.

On doc/orm files for Object Reference Manual, are located, mainly picture files as the text reside in the wbl components. In doc/web are menus and frames for the documentation page.

Tools

Tool components only exists in the rt module, $pwre_croot/src/tools, and contains different tools to build a Proview release. Below tools/exe there are various exe files, for example to convert msg-files. In tools/bld generic makefiles are located, in tools/pkg build files for installation packages and in tools/pwre the script to build a Proview release, pwre.

Overview

Appendix A contains an overview of the component of the various modules.

Flavor

The level below component is denoted flavor. Normally this is a src-directory containg the source code for the component, but in some cases the src-directory is divided in directories for different windowing systems (motif or gtk), or different languages.

gtk/motif

Originally the the window interface was developed for Motif on OpenVMS, but from V4.3 gtk was implemented parallel to Motif. The design makes the different interfaces independent of each other, and it is possible to only build for one of them. It is also fairly easy to implement other windowing systems.

This division in motif and gtk is found in the exe and lib components.

On the lib-components there is a base class with common code in the src catalog. In the gtk and motif catalog, resides a subclass with code that is specific for gtk or motif. If we look at $pwre_croot/xtt/lib/xtt, that contains code for the operator environment, on the src-catalog the files xtt_op.cpp is located, containing the class XttOp. On the gtk-catalog the file xtt_op_gtk.cpp is found, containing the class XttOpGtk, a subclass to XttOp, and on in the motif catalog, the file xtt_op_motif.cpp is found with the class XttOpMotif, also a subclass to XttOp.

In the exe component, there is a corresponding division. From the src catalog a generic exe-file is generated, that starts the gtk or motif version dependent on the option -f. The gtk catalog generates the exe-file $pwr_exe/rt_xtt_gtk and the motif-catalog $pwr_exe/rt_xtt_motif. If rt_xtt is started with -f gtk, the gtk-version is started, and with -f motif the motif-version is started. In gtk/os_linux there is a build file, link_rule.mk, that contains the link command to build the gtk version, and in gtk/os_linux/hw_x86 there is a makefile to link the gtk version. Corresponding catalogs are found below the motif directory.

Language

The flavor level is also used to divide into different language versions. One example is $pwre_croot/xtt/mmi/xtt that has the directories src, en_us, sv_se, de_de and fr_fr. The catalogs contains laguage specific files for texts in menus, windows and object graphs for English, Swedish, German and French. When building, the files in the catalog xtt/en_us are copied to the catalog $pwr_exe/en_us, and the files in xtt/sv_se to $pwr_exe/sv_se etc. When the operator environment is started with a specific language, the translation files are fetched from the corresponding catalog below $pwr_exe.

Os

Below the flavor level are directories for different operative systems, e.g os_linux, os_macos and os_freebsd. On this level, files that are specific for that operating system is located, for example c-files that contains several system calls or shell-script files. Also files involved in the build environment are located here, for example the make-files are often found on the os-level, or even lower down in the hw level. Files that are common for several operating system, as for example generic make-files, are stored in the catalog os_templ, and are copied to the catalogs for specific operating systems when the build tree is configured. If all files are generic, the operative system catalog is created as a hidden catalog, e.g .os_linux, at configuration.

Hw

The lowest level is the hardware level, hw. Below the os_linux catalog, the hardware catalogs hw_x86, hw_x86_64 and hw_arm is found, for linux on x86 , 64-bit linux on x86 and arm architecture respectively. On this level, what differs are often switches to linkers and compilers. Generally there is a makefile on this level to build a component for a specific operative system and hardware. Also here common files are stored in a hw_templ catalog, and if only common files exist on this level for a component, a hidden catalog, e.g. .hw_x86 are created at configuration.

Build tree

The build tree is a hierarchy of catalogs created when building a Proview release. Here the result of the building is placed, i.e. the exe-files, archives, graphs etc that is necessary to configure and run a Proview system. Chosen parts of the build tree are collected into installation packages for development, process, operator and storage stations, but it is also possible to link project directly to the build tree.

The build tree consists of 5 levels. The environment variable $pwre_broot points to the root of the build tree, and below this there is one level for operating stytem, and below this further on for hardware, e.g.

$pwre_broot/os_linux/hw_x86

On the next level there is one directory for each module, where the components of the module are stored. There is also an exp directory where the module directories are merges together to a common distribution. Finally there is also a bld directory containing build-files of temporary nature, that are needed for the building, but not required in the distribution.

Behind the design of separate module directories is the idea that modules can have separate installation packages and that a Proview installation in this way could be more scalable. This is though not yet implemented in any module.

Module

Every module has its own catalog structure i the build tree, where files that is to be included in the distribution is stored. For example, the catalogs for the rt module is found under

$pwre_broot/os_linux/hw_x86/rt

Below this catalog the catalogs exe, lib, obj, load, inc, doc, db and cnf are located. A corresponding catalog structure is also found for the other modules, and for the exp catalog.

Exe

On the exe catalog, exe-files that are created when linking an exe-component, is placed, e.g. rt_ini that is generated from the exe component $pwre_croot/src/exe/rt_ini.

Other files that are copied to the exe-catalog are shell scrips, object graphs, subgraps etc.

The exe catalog has language dependent subdirectories, e.g. en_us, sv_se and de_de, for English, Swedish and German. Here resides also helptext files and translation files for different languages.

Lib

On the lib catalog resides archives generated when building a lib component, e.g. libpwr_rt.a that is generated by the lib component $pwre_croot/src/lib/rt. Also java archives generated by jpwr components are placed in lib, e.g. pwr_rt.jar.

Obj

On the obj catalog various o-files are found, generated when compiling c and c++ files.

Load

The load catalog contains loadfiles generated by wbl components, for example pwrs.dbs, generated by the wbl-component $pwre_croot/src/wbl/pwrs, that contains the classvolume pwrs with system classes. On load you will also find flw-files, copied from wbl-components, that are used by plc-trace.

Inc

The inc catalog contains include files from lib components, and include files that are generated from wbl components with c-structs and c++ classes for classvolumes, e.g. pwr_abbclasses.h and pwr_abbclasses.hpp.

Doc

The doc catalog contains the complete documentation for a Proview release. Note that all modules uses the doc catalog below exp, and that the doc catalogs in the modules are not used. Doc contains language dependent subdirectories, en_us and sv_se, where the language specific files are found. If we take a look at en_us, we find the documentation homepage, index.html that is copied from $pwre_croot/src/doc/web/en_us. We also find manuals in pdf and html format generated from $pwre_croot/src/doc/man/en_us. On the subdirectory orm resides the Object Reference Manual that mainly is generated from the wbl components for the classvolumes. The catalog doc/prm contains Programmer's Reference Manual, generated from the lib/rt and lib/co components by doxygen.

Exp

Beside the module directories in the build tree, you find the exp directory, that is a merge of the different modules, and that constitutes a Proview distribution. Exp is for export, and it is this part of the build tree that is exported in a complete Proview release. The exp catalog contains a similar catalog structure as each module catalog, you will find the subdirectories exp, lib, load, inc etc. When merging the modules to the exp directories the content of a module is basically copied to the exp catalog. But there are some cases where a simple copy is not enough. Some lib-components are represented in several modules, rt and wb, and here the archives are merged to a common archive, exp/lib/libpwr_rt.a and exp/lib/libpwr_wb.a. Some exe-files contains methods for, for example I/O handling and popup menues in the operator and development environment, that derives from different modules, and these have to be linked in a certain way to embrace all the methods. This goes for the exe-components $pwre_croot/wb/exe/wb, $pwre_croot/xtt/exe/rt_xtt and $pwre_croot/exe/rt_io_comm.

It is possible to link projects to the release on the exp-catalog, where you can run both the development, runtime, operator and storage environment. You then define the exp-catalog as a version under Base in the ProjectList and attach the projects to this version.

Bld

Beside the exp and module catalogs in the build tree, a bld catalog is located. This contains files that are used at the build, but don't need to included in the release. The subdirectories reflects the

different components and are common for all modules.

Below bld/lib there are catalogs for all lib components, e.g. bld/lib/rt. Here o-files are located, generated at compilation of c and c++ files, before they are inserted into archives. You vill also find some .d files which are dependency files for the c-files.

Below bld/exe there are catalogs for all exe components, e.g bld/exe/rt_ini. Here you will find o-files, that are linked to exe-files, and d-files with include file dependencies.

Below bld/jpwr there are catalogs for jpwr components, e.g. bld/jpwr/rt. Class-files, generated at the java compilation, are located here, before they are inserted into java archives.

Below bld/msg here are catalogs for msg components. The cmsg-files contains the text for different status codes. Below bld/wbl resides dependency files for wbl components.

In bld/pkg the installation packages for pwr47, pwrrt, pwrsev and pwrdemo, are placed. These are built from pkg components below tools/pkg where the build files for installation packages are located.

Build

Building Proview implies to, from the source code tree, generate a build tree and a Proview release the programs, archives, loadfiles, manuals etc. that are needed to install and run development, process, operator and storage stations.

The build is executed with the pwre command. First you create a build environment, by stating the root of the source tree and build tree. The environment is stored in a file, in which you can store several different environments. In this way its easy to attach to an environment, and to shift between different environments.

Normally you build a complete release, but it is also possible to build only the runtime code, i.e. the rt module, or the runtime and HMI code, i.e. the modules rt and xtt. In this case though, some platforms independent files has to be imported from a complete release, by defining an import root.

Pwre is located in the source tree in the catalog $pwre_croot/tools/pwre. On linux, pwre is a perl script, pwre.pl, located on the subdirectory src/os_linux.

Preparation

For a complete build these packages has to be installed:

```
make
flex
gcc
g++
libgtk2.0-dev
cpp
libasound2-dev
libdb4.6-dev
libdb4.6++-dev
doxygen
libmysql++-dev (optional)
```

java: Download jdk-6u10-linux-i586.bin from java.sun.com. Define the environment variable jdk to the path where the package is extracted (e.g. /usr/local) and put $jdk/bin in the PATH.

```
export jdk=/usr/local/jdk1.6.0_10
export PATH=$PATH:$jdk/bin
```

antlr: Download and build source package antlr-2.7.7 from www.antlr.org. On fedora configure with './configure --disable-csharp'.

Hints:

–You may have to rename /usr/bin/jikes and /usr/bin/java temporary during configure.

–Include <strings.h> in lib/cpp/antrl/CharScanner.hpp

There has to be a valid display when building proview.

Pwre

Before starting pwre, two env variables has to be defined. One that points to the catalog of the pwre script, $pwre_bin, and one that states the name of the file where the environments are stored, $pwre_env_db. You also have to execute a setup script, $pwre_bin/pwre_function. In the example below, the source code is located in /data0/x4-7-0/pwr and the database is placed on the home directory.

```
> export pwre_env_db=~/pwre_en_db
> export pwre_bin=/data0/x4-7-0/pwr/src/tools/pwre/src/os_linux
> source $pwre_bin/pwre_function
```

You then create the directory where the build tree is to be placed, in this example /data0/x4-7-0/rls.

```
> mkdir /data0/x4-7-0/rls
```

Now we can create the environment named x470

```
> pwre add x470
Source root [] ? /data0/x4-7-0/pwr/src
Import root [] ?
Build root [] ? /data0/x4-7-0/rls
Build type [dbg] ?
OS [linux] ?
Hardware [x86] ?
Description [] ? Version V4.7.0
```

Note that in 'Source root' the root of the rt-module is stated, not the actual source root which is /data0/x4-7-0/pwr.

The command 'pwre list' shows stored environments.

```
> pwre list
-- Defined environments:
x470              Version V4.7.0
```

With the command 'pwre init' you attach an environment, that is you define a number of env variables that point to the source tree and the build tree. This has to be done in every session where you work with the environment.

```
> pwre init x470
```

Here are some useable env variables that is defined

$pwre_croot	The source root (/data0/x4-7-0/pwr)

$pwre_sroot	The source root for the current module
$pwre_broot	The build root (/data0/x4-7-0/rls).
$pwr_eexe	The common exe directory in the build tree ($pwre_broot/os_linux/hw_x86/exp/exe).
$pwre_elib	The common lib directory in the build tree ($pwre_broot/os_linux/hw_x86/exp/lib)
$pwr_exe	The exe directory for the current module.
$pwr_lib	The lib directory for the current module.

Create directories in the build tree

The next step is to create all the directories of the build tree

```
> pwre create_all_modules
```

Configure

`pwre create_all_modules` calls a configure function that examines the environment and creates a file configuration file `$pwre_broot/pwre_'platform'.cnf` to adapt the build to the current installation. If you install additional packages you should run the configure function again to update the configuration file.

```
> pwre configure
```

Complete build

This command build all modules, i.e. perform a complete build of Proview.

```
> pwre build_all_modules
```

By default this command will build for flavor gtk, if you want to build for motif instead, motif is added as argument

```
> pwre build_all_modules motif
```

Build a component

When working with development of a part of Proview, you often make changes that affects one or a couple of components. To build an individual component you first have to set up the module.

```
> pwre module 'module'
```

You then build the component with the command

```
> pwre build 'type' 'component' 'flavor'
```

When the building is performed, the result is stored in the build tree for the current module. This now has to be merged with the exp directory in the build tree

```
> pwre merge
```

Example

If a modification is made in lib/wb/src in the wb module, the command is:

```
> pwre module wb
> pwre build lib wb src
> pwre merge
```

Phase

The build is divided in four phases: init, copy, lib and exe.

The basic idea is that the init phase creates directories and archives needed for the build, the copy phase copies include files and other files, the lib phase compiles c, c++ and java files, and finally the exe phase links the exe files. This goes for lib and exe components, for other components the phases are use somewhat different.

The phase can be specified in the pwre command when building a component as the fifth argument, for example

```
> pwre build lib wb src copy
```

where the last copy is the phase. If the phase is left out, all four phases are executed.

Below follows a description of what is executed in the phases for different components.

Type	*Phase*	*Description*
lib	init	Creates a build directory with the component name in the bld/lib directory in the build tree.
	copy	Converts pdr and xdr files to h files, and copies h and hpp files to $pwr_einc.
	lib	Compiles all c and cpp files that has the component name as prefix. The resultant object modules are store in the build directory. Creates an archive on $pwr_elib and inserts the object modules.
	exe	-
exe	init	Creates a build directory with the component name in the bld/exe directory in the build tree.
	copy	Copies all h and hpp files with the component name as prefix to $pwr_einc.
	lib	Compiles all c and cpp files. The resultant object modules are store in the build directory.
	exe	Links with the link command defined in the link_rule.mk file. Places the resultant executable on $pwr_eexe.
wbl	init	-
	copy	Create includfiles with c structs and c++ classes for all classes in the volume. These files pwr_'volume'classes.h and pwr_'volume'classes.hpp are placed on $pwr_einc. Also copies pwg and pwsg files to $pwr_eexe and flw files to $pwr_eload.
	lib	Creates a dbs file on $pwr_eload, 'volume'.dbs.
	exe	Creates documentation for the volume, help-files, html-files, postscript and pdf-files.

msg	init	Creates a build directory with the component name in the bld/msg directory in the build tree.
	copy	Generates h-files for the status codes on $pwr_einc, and c-files (cmsg) with the text on the build directory.
	lib	Compiles the cmsg files.
	exe	-
exp	init	-
	copy	Copies different files, h, hpp, pwg, sh, pwr_com etc.
	lib	Compiles c and cpp files.
	exe	-
mmi	init	-
	copy	Copies pwg, pwsg and png files to $pwr_eexe. Compiles uil-files to uid-files on $pwr_eexe.
	lib	-
	exe	-
jpwr	init	Creates a build directory with the component name in the bld/jpwr directory in the build tree.
	copy	-
	lib	Compiles java files to class-files on the build directory. Creates a java archive on $pwr_elib and inserts classes and gif-files.
	exe	-

Method dependent exe components

There are three components that has to be built with a special command to bring forward various types of methods at the build. This concerns wb/exe/wb, xtt/exe/rt_xtt and src/exe/rt_io_comm. Those are built with the command 'pwre method_build'.

```
> pwre method_build wb gtk
> pwre method_build rt_xtt gtk
> pwre method_build rt_io_comm
```

Build with an import root

From V4.7.0.

When building a common release on different platforms, the version of the loadfiles should be the same on all the platforms. This can be achieved by defining an import root. The idea is to build the dbs-files on one platform, and define import roots on the other, and the copy the dbs-files form the import root, instead of building them.

When creating the environment with 'pwre add' the import root is stated. In the exemple below it resides on a remote node, pwrdeb.

```
> pwre show
```

```
--
-- Environment    : x470_64
-- Module.........: rt
-- Source root....: /data0/x4-7-0/pwr/src
-- Import root....: pwr@pwrdeb:/data0/x4-7-0/rls/os_linux/hw_x86
-- Build root.....: /data0/x4-7-0_rt/rls
-- Build type.....: dbg
-- OS.............: linux
-- Hardware.......: x86
-- Description....: X4.7.0 on 64 bit debian
```

Create the build tree directories

```
> pwre create_all_modules
```

Import the dbs files from the import root

```
> pwre import dbs
```

Build all the modules

```
> pwre build_all_modules
```

Build for embedded platforms

When building for embedded systems with a cross compiler, it's not possible to build a complete release with the development environment. Instead some files generated from the development environment is imported from a complete release. In pwre the path to this import release is stated, and with the 'pwre import' command files are imported.

A cross compiler has to be defined with the environment variables pwre_cc, pwre_cxx and pwre_ar that should point at the c, c++ compiler and the archive program ar.

For the build, some programs has to be executed and pwre_host_exe should point to the exe directory of an release of the development platform, usually the same release as the import root.

We begin with defining the pwre links to the compiler tools . In the example we are building for Raspberry Pi.

```
export pwre_cc=/usr/local/rpi/raspbian/bin/arm-linux-gnueabihf-gcc
export pwre_cxx=/usr/local/rpi/raspbian/bin/arm-linux-gnueabihf-g++
export pwre_ar=/usr/local/rpi/raspbian/bin/arm-linux-gnueabihf-ar
```

Define a link to the exe directory of the host release

```
export pwre_host_exe=/data1/x5-0-0/rls/os_linux/hw_x86/exp/exe
```

Create an pwre environment for the rpi release with hardware arm

```
pwre add x500rpi
Source root? /data0/x5-0-0/pwr/src
Import root? /data0/x5-0-0/rls/os_linux/hw_x86
Build root? /data0/x5-0-0/rls
Build type?
OS? linux
Hardware? arm
```

Build the arm release

```
pwre init x500rpi
mkdir $pwre_broot
pwre configure --ebuild
pwre create_all_modules
pwre import rt
pwre import java
pwre ebuild rt
```

In the above example the embedded release root is common with the host release and probably already defined int the project list. If another root is used it should be given a version name in the project list, $pwra_db/pwr_projectlist.dat, eg

```
%base X5.0.0rpi     /data0/x5-0-0/rls
```

As default this will build the runtime part of all modules. It is possible to disable the build of not needed modules by editing the ebuild.dat file on $pwre_bin.

```
bcomp           1
java            1
remote          1
nmps            1
sev             1
opc             1
profibus        1
otherio         1
ssabox          1
tlog            1
othermanu       1
abb             1
siemens         1
klocknermoeller 1
inor            1
telemecanique   1
```

If 1 is exchanged to 0 for a module, this module will not be built. Note there can be a dependency between modules. The seimens and abb modules are, for example, dependent on the profibus module.

Configure an embedded project

Normally there is also a need to build a project on the runtime only release. The project has to point at the complete release, because that's where the development environment is present, but the build command for the node and plcprogram has to be directed to the runtime only release. To do this you set the operating system for the node, and for the root volume to CustomBuild, and create a CustomBuild object below the NodeConfig object for the node in the directory volume. In the CustomBuild object the cross compiler tools are stated.

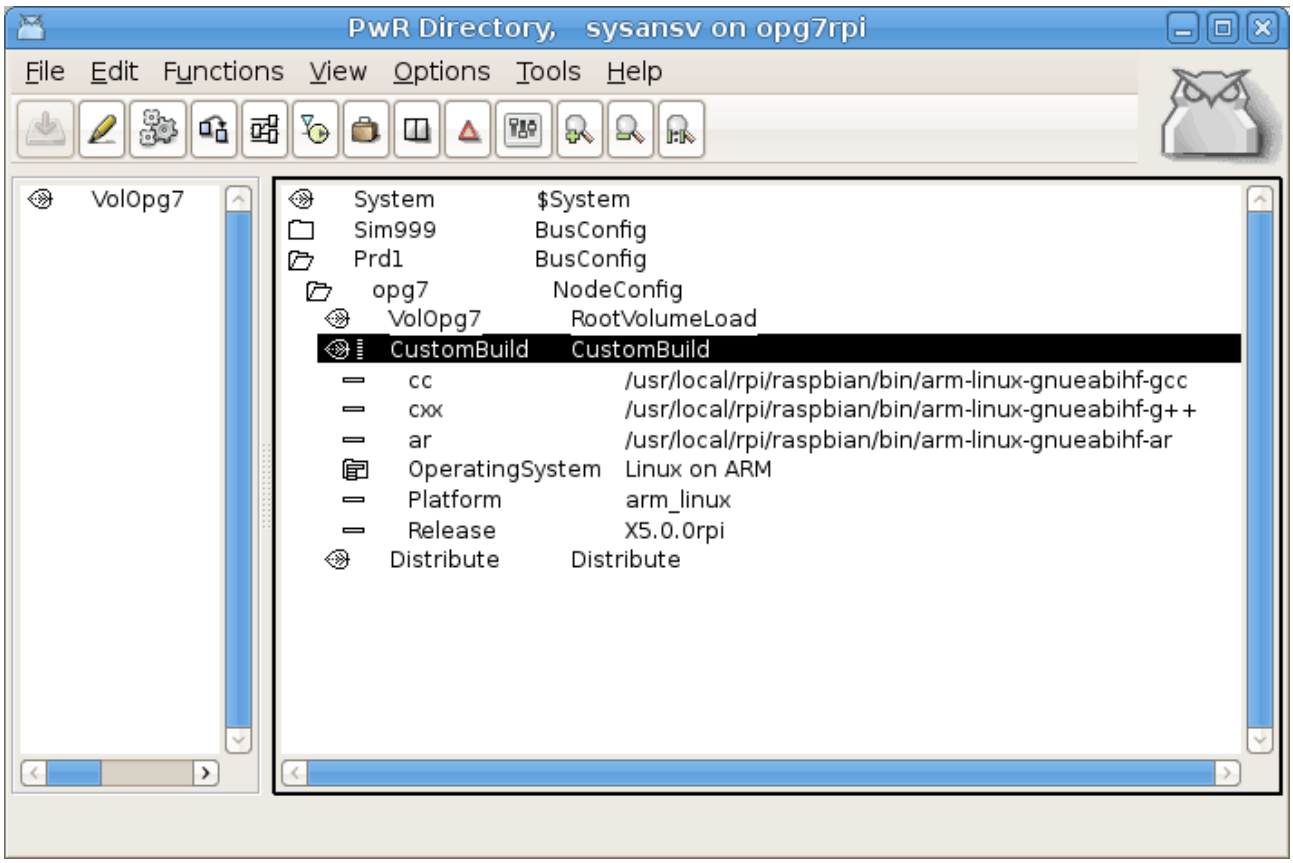

Fig CustomBuild object defining the embedded environment

Build operator environment only

From V4.7.0

Building an release with only the operator environment is made in a similar way. As this includes the runtime environment, runtime is build first as described in the previous section. Then the operator is build with

```
> pwre import op
> pwre ebuild op
```

Appendix A

Component overview

Module rt

Component		*Description*
lib	co	Contains common functions for runtime and development environment, e.g. time functions, command line interpreter, handling of different languages, xml parser.
	rt	This is the basic runtime library with functions for the runtime environment, e.g. the realtime database, handling of alarms and events, plc, io handling, subscriptions etc.
	dtt	Contains the system pictures in rt_rtt.
	msg_dummy	?
msg	co	Msg-files for lib/co.
	rt	Msg-files for lib/rt och exe/rt_*.
	flow	Msg-files for xtt/lib/flow.
	glow	Msg-files for xtt/lib/glow.
	ge	Msg-files for xtt/lib/ge.
	rs	Msg-files for the modules remote, nmps, ssabox och tlog.
	wb	Msg-files for wb/lib/wb.
wbl	pwrs	Classvolume pwrs, system classes.
	pwrb	Classvolume pwrb, base classes.
	rt	Shared volume rt. Contains sound objects.
	wb	WorkBenchVolume. Contains list descriptors.
mmi	co	Common picture files.
exp	com	Shell scripts and command files.
	inc	Include-files for basic types, classes and definitions.
	rt	Various files.
	stdsoap2	Files for gsoap, used by the status server.
exe	co_convert	Program to generate h-files and convert between different formats. Generates h and hpp files for class volumes from wb_load files, converts from xtt-help files to pdf, postscript and html etc.

	co_merge	Used by pwre when handling modules.
	pwr_user	Command interface to the user database.
	rt_bck	Backup of objects in the realtime database.
	rt_bck_dump	Create a dump from the backup file.
	rt_elog	The event log server. Stores event and alarms in a database.
	rt_emon	Event monitor. Handles alarms and events.
	rt_fast	Handles fast curves.
	rt_ini	The startup program for the runtime environment. Creates the realtime database and starts the system processes, plcprogram and applications. Also handles the consol logging.
	rt_io_comm	Process for I/O handling of units that are not handled by the plc program.
	rt_linksup	Supervision of links to other Proview nodes.
	rt_mozilla	Program that starts a web browser.
	rt_neth	Nethandler. Provides other nodes with information about the realtime database.
	rt_neth_acp	Handles links with other Proview nodes.
	rt_prio	Sets priority on system and application processes.
	rt_qmon	Qcom monitor. Handles communication with other nodes.
	rt_rtt	Tool to examine the system and the realtime database from a terminal window.
	rt_sevhistmon	Collects history data and sends to storage stations.
	rt_statussrv	Provides information about system status for the Supervision Central.
	rt_sysmon	System monitor. Supervises the system.
	rt_tmon	Timer monitor. Sends subscriptions to operator stations.
	rt_trend	Handles trend curves.
	wb_rtt	An editor for rt_rtt pictures.
doc	man	Contains manuals and help texts.
	web	Web-files for the documentation home page.
	orm	Files for the Object Reference Manual.
	prm	Programmer's Reference Manual.
	dox	Doxygen definitions.
tools	pwre	Build script to build Proview from sources.
	bld	Makefiles to build Proview.
	pkg	Files to build installation packages.
	exe	Various programs to create msg-files etc.

Module Xtt

Component		*Description*
lib	cow	Common graphical functions and window for messages, runtime monitor, status monitor and helptext viewer.
	flow	Flowchart editor (flow) used by the plc editor and plc trace. Also a browser used for example in the navigator.
	glow	Graphical package for process graphics and the Ge editor.
	ge	The Ge editor, process graphics.
	xtt	Functions in the operator environment, navigator, alarm windows, trends, fast curves and history curves, operator window etc.
exe	rt_xtt	The Proview operator environment.
	wb_ge	Separate program for the ge editor.
	pwr_rtmon	Runtime monitor. Program to start/stop Proview runtime.
	rt_statusmon	Supervision central. Program to supervise Proview nodes.
	co_help	Separate program to view help texts.
mmi	ge	Picture files for the Ge editor.
	xtt	Picture files for xtt.
	sis	Subgraphs for SIS.
	ssg	Subgraphs for SSG.
exp	ge	Object graphs and type graphs.
	inc	Include files for bitmaps used as icons in navigator and palettes.

Module Wb

Component		*Description*
lib	wb	The main library for the development environment. Contains the development database, the configurator, the plc editor, the spreadsheet editor etc.
exe	wb	The main development tool of Proview.
	wb_cmd	Command line and script interface to the development database.
	wb_ldlist	Program to examine the version of a dbs-file.
	wb_upgrade	Program sometime used by the project upgrade procedure.
mmi	wb	Picture files for the development environment.
exp	wb	Various files.
	com	Command files and shell scripts.

Module Remote

Component	*Description*

lib	remote	Common functions for remote.
exe	rs_remotehandler	Main program for the remote function.
	rs_remote_3964r	Communication to a remote system using Siemens 3964r on a serial line.
	rs_remote_alcm	Communication to a remote system using the ALCM protocol.
	rs_remote_modbus	Communication to a remote system using Modbus on a serial line.
	rs_remote_mq	Communication trough a message queue using BEA MessageQ.
	rs_remote_rk512	Communication to a remote system using rk512.
	rs_remote_serial	Communication to a remote system using a serial line.
	rs_remote_tcpip	Communication to a remote system using the TCP/ip protocol.
	rs_remote_udp	Communication to a remote system using the UDP/ip protocol.
	rs_remote_logg	Program to log communication on file.
	remote_pvd_pwrcli	Provider program to mount a Proview system as an extern volume.
	remote_pvd_pwrsrv	Server program for remote_pvd_pwrcli.
wbl	remote	The Remote classvolume.

Module Nmps

Component		*Description*
lib	nmps	Contains code for function objects and application interface for Nmps.
exe	rs_nmps_bck	Backup of cells and data objects.
	rs_nmps_bck_dump	Program to examine a backup file.
wbl	nmps	The NMps classvolume.

Module Profibus

Component		*Description*
lib	rt	Contains I/O methods for profibus and profinet.
	cow	Contains the profibus configurator and the profinet configurator.
	xtt	Xtt-method to open the profibus configurator in rt_xtt.
	wb	Wb-methods to open the profibus configurator and profinet configurator.
exe	profinet_viewer	Program to show connected devices on the profinet circuit, and to set name and adress on the devices.
	pn_get_deviceid	Program to extract ProductFamily and TextInfo from gsdml files and generate a database for the profinet configurator.
wbl	mcomp	The Profibus classvolume.
mmi	mcomp	Object graphs.
	pb	Uil file for the motif version of the profibus configurator.

exp	gsd	Contains gsd-filer, e.i. descriptions files for profibus slaves.
	rt	Contains help texts.

Module Opc

Component		*Description*
lib	opc	Common functions and the gsoap interface.
exe	opc_provider	Opc client, implemented as an external volume.
	opc_server	Opc server.
wbl	mcomp	The Opc classvolume.
exp	mcomp	Object graphs and type graphs.

Module Java

Component		*Description*
jpwr	rt	Java runtime interface.
	rt_client	Archive to execute java remote and get info from the realtime database via socket communication.
	jop	Operator interface in java.
	jopc	Object graphs in java.
	beans	Components to build java graphics in for example JBuilder or other IDE
	bcomp	Object graphs for Basecomponent objects.
	abb	Object graphs for ABB objects.
exe	jpwr_rt_gdh	Java native for gdh, qcom, errh and mh classes.

Module Otherio

Component		*Description*
lib	rt	I/O methods for various I/O units.
	usbio_dummy	Archive to be able to link the plc without installing MotionControl USBIO.
wbl	mcomp	The OtherIO classvolume.
mmi	mcomp	Object graphs.
exp	rt	Include-files for external archives.

Module Bcomp

Component		*Description*
lib	rt	Code for plc function objects.
	wb	Wb methods for various classes.
wbl	bcomp	The classvolume BaseComponent.
mmi	bcomp	Object graphs and graphical symbols.

doc	orm	Pictures to Object Reference Manual.

Module Othermanu

Component		*Description*
wbl	mcomp	The classvolume OtherManufacturer.
mmi	mcomp	Object graphs and graphical symbols.
doc	dsh	Datasheet for various components.

Modules ABB, Siemens, Inor etc

These modules for various manufacturers are designed in a similar way.

Component		*Description*
wbl	mcomp	The classvolume for the module.
lib	rt	Possible I/O methods.
	wb	Possible wb methods.
mmi	mcomp	Object graphs and graphical symbols.
doc	dsh	Datasheets for various components.
	orm	Pictures to Object Reference Manual.