qef Technical Overview

The qef environment consists of qef, its supporting programs, a graphical user interface and documentation. This introduces some of the more important features and their benefits.

Basic *qef* Architecture

qef is primarily a driver that manages three stages, which are:

1) Configuration and control variable database preparation
2) Script preparation
3) Back end interpretation.

This architecture affords a great deal of flexibility and power. ***qef*** may be used to manage a wide range of processing in a single and consistent user interface. The ability to use arbitrary script preparation and arbitrary back-ends supports the choice of "the right tool for the job." Script preparation also allows for dynamic run-time modification of the job to be done.

The *qvrs* Configuration Parameters System

qef uses a separate program, ***qvrs***, to manage the configuration and control parameters, variables, search paths, tool names, etc. Usually ***qvrs*** is invoked to build variable/value database for other tools such as ***qef***. It also provides debugging facilities and simple run-time or temporary over-rides. Part of the basic ***qvrs*** facility is a mechanism to incorporate a user's build-specific configuration file.

The ***qvrs*** approach eliminates most of the problems involved in configuration settings and control. The variables provide for specification of platform, project, sub-project, build-config, user and directory specific settings. Using a simple language, this versatility, extensively documented, will effectively insure an easily tailored configuration. This language offers a rich set of functions providing system tests, states of the system, variable settings and retrieving a variety of information. To facilitate testing of variables, ***qvrs*** has a number of flags that output a variety of information about variables and their settings.

Script Preparation

Script preparation is done at run-time. This can be a simple specification of a file to be processed, or the specification of a shell command to be run to prepare the script. The most commonly used approaches are to use ***qefdirs*** to translate a list of directories and their supported constructions into a make-like script, or to use ***sls*** to generate a list of source files for the directory and ***qsg*** to translate that list and simple commands into input for a ***qef*** provided back-end.

The ability to specify an arbitrary process to generate the input provides great flexibility and allows one to reduce the required facilities within a directory to their simplest form. This also encourages the provision of all required processing within a single and consistent interface — "All one ever says is '***qef***'."

qsg — The Main *qef* Script Generator

qsg, the primary ***qef*** script generator, is a programming system specifically designed to generate input to other processes. Its major purpose is to handle statements of the following form:

```
commands @argv              # process all the sources in the current
                            # directory as single module files
library -n Name -v @argv    # create a library called Name
                            # from the sources in the current directory,
                            with a version file
```

The scripts c*ommands* and *library* in the above examples are scripts in the standard ***qsg*** archive.

They will generate the make-like script to process all the argument files, with complete and comprehensive targets and facilities. The scripts themselves deal with host-specific problems. Facilities exist to trivially add support for new languages and file types (based on suffixes).

In 1200 qef files at three different sites, the average length was 3 lines of qsg input. Many were one-liners, and those same qef files would be used for all systems; yet the generated build-scripts would contain complete and comprehensive controls, allowing the user to select the construction of any specific target or intermediate file and supported variations (i.e., if you have purify, facilities to build the purify objects are created). Compare this with the huge incomplete static makefiles, sometimes replicated for each different host platform.

Script Macro Processing

Normally, ***qef*** invokes a built-in C-like macro processor, henceforth referred to as ***qefpp***, to process the generated scripts before output to the back-end. This is usually to perform tool-name substitution, source path resolution, insertion of flags into the build-recipes, and incorporation of dependency lists (see ***incls***).

qefpp greatly eases the task of creating a script (possibly via a script generator), because the generator or user can use macros for paths, tools, and flags, the values of which are usually extracted from the ***qvrs*** database.

Separation of Source and Object Trees

qef's run-time generation and processing of scripts facilitates (and encourages) the total separation of source and object cheaply and effectively: just naming a source tree to be included in a ***qvrs*** file is required (and the tools are provided to help). The script generators, macro processors, and many of the other tools search the source path for files either on request or automatically.

The benefits of this separation are many. All users may share a single master (read-only to most) tree of sources. Developers have their own trees to contain the files they are changing or creating, thereby eliminating the problems of developers interfering with each other or production. Builds are done in separate directories, thereby facilitating building using the same sources for multiple configurations. Note: Setting up a separate tree is a single command that simply creates a couple of ***qvrs*** files and creates the required or requested sub-directories.

qdsrv – The Universal *qef* Directory Database

qdsrv is a network server program that maintains a database of project trees. ***qdadd*** adds new entries to this database and ***qds*** extracts selected entries, either by index or attributes. Other programs exist to manage and check the entries and to bind entries to a user's session. The creation of a new tree automatically adds that tree's entry to the database.

The ubiquitous and major problem within any organization as to where a project's sources and developments are to be found is solved! A user does not need to know actual pathnames of trees. A one-line shell alias or function can be created that allows the user to ***chdir*** to a directory named by specification of the desired tree's owner, name, release, host, configuration, and/or type.

incls – Dynamic Dependency Tracking

Usually ***qefpp*** includes the running of a command to insert the list of source implied dependencies (e.g., the # include statement in C) into the script. The program ***incls*** is used to do this. ***incls*** is extremely fast and maintains a database to avoid rescanning files that have not changed. It currently handles eight different languages varying from C to TeX to SmartWare printer description files. Facilities are provided to add new languages. ***incls*** also supports processing individual files or lists of files of arbitrary types, providing a variety of output formats. The new consistency engine under development incorporates lazy evaluation of dependencies.

The automatic inclusion of **all** dependencies is fairly cheap and painless. On 450 ‘C’ source files, ***incls*** took 1/5 the time of X11’s ***makedepend*** when it had to scan all its input files. The second time **incls** was 25 times faster. Larger comparisons were not possible, as ***makedepend*** has built-in limits. Also note that ***makedepend*** is limited to C and a single output format. Even if ***incls*** is more costly, it is worth it. Its pedantic creation of **all** dependencies eliminates a large amount of expensive programmer time (i.e., hunting for bugs caused by a header file version skew).

Library Naming and Mapping

Another of ***qefpp’s*** major functions is to find and resolve libraries for programs. Using various ***qvrs*** variables and information extracted from the relevant source files, ***qefpp*** searches for the libraries to be linked with a program to ensure a complete dependency list. It also provides the appropriate list to the linking commands. Mechanisms are provided for specification of overrides, remappings of the library symbolic names and the use of static or shared libraries for individual or all libraries for either individual or all programs. A separate tool, ***libs***, is provided to test the mappings and searches.

The mechanisms provided allow the specification of a program’s libraries using a comment embedded in the source file for that program. For example:

```
/* LIBS: -lXaw -lXmu -lXt -lXext -lX11
 */
```

The various ***qvrs*** library mapping variables can then be used to resolve any necessary adjustments for the configuration. Usually most such adjustments are made universally for a system or project. This simple specification can be converted into the appropriate path names and/or flags in the required forms, yet easy mechanisms to control these mappings are available via ***qvrs*** settings. For example:

```
set ExtraLibs -lplumber         # add -lplumber to every program
set LibMap[-lX11] -lX11 -lbsd   # add -lbsd to any use of -lX11
set LibStatic[-lXaw] X Y*       # static -lXaw library used for programs X and Y*
```

Version System Interface

This facility provides a system-independent interface to be used within qef to provide basic source code control functions. More sophisticated commands can be invoked by the user via the command line window or by invoking the version control system directly. Each version system will be required to support the following:

`co`	check out
`ci`	check in
`diff`	show differences between gfile and vfile
`vdiff`	show differences graphically – using provided xvdiff
`tell`	tell me about editing activity within current directory
`new`	create a new administration file

qef's version control interface provides complete implementations for SCCS and RCS. Other version systems with a command line interface can be easily added to qef. To create a new interface, copy an existing interface file for RCS and modify it for a new version control system.

Back-Ends

The ***qef*** product contains a number of programs built specifically for use as back-end interpreters. ***qsh*** is a limited shell command interpreter that provides suitable command echoing, selected ignoring of the exit status, and a gentle halt. ***mimk*** is a make-like process that provides multiple parallel execution of recipes, a history, extended consistency rules (e.g., recipes applied if recipe or dependency lists have changed), and a gentle halt. As part of its dependency mechanism, ***mimk*** allows intermediate files to be missing but still considers the target to be up-to-date. ***qmk***, a ***mimk*** extension under development, provides lazy dependency evaluation, extended history mechanisms, and more controls over execution ordering.

The parallel execution has substantial performance benefits. The extended consistency mechanism goes a long way towards guaranteeing source/product consistency. The ability to halt a project reliably with a trivial mechanism should be recognized as being valuable by any experienced reader.

The Graphical User Interface (GUI)

qef has a graphical user interface to provide tree navigation, access to the user documentation, build controls and monitoring, access to the project database, file management, an interface to the arbitrary version system, etc.

The GUI greatly reduces the learning curve and understanding of the system. This results in accelerated implementation and acceptance of the system by the development teams.

Additional Tools & Utilities

A number of tools are provided to deal with aspects of the construction task.

instal is a paranoid (checks everything it can) installation process that: maintains an audit trail; retrieves installation flags from the ***qvrs*** database; provides option to record chown, chmod, or chgrp errors rather than abort; optionally adds DOS CR-LF to the installed files; suppresses installation if the file's content will not change; creates necessary directories; and performs some work-a-rounds to deal with some systems' bugs.

mkvernum is a program used to create version files or strings for embedding in other files or projects.

One of the formatting controls is to include a count that is incremented on each use, thereby providing a unique identifier for the version string. A number of utilities provide options to embed ***mkvernum*** strings in their produced files.

arupdate is a sophisticated interface to the archiver. Its primary feature is that one specifies the list of all the files it should contain, rather than those to be added as is the case with the standard tool ***ar***. Thus ***arupdate*** can remove files that should not be in the archive. It can embed ***mkvernum*** generated objects in the database, include all or selected parts of other libraries, and accept file listing objects to be added to the library. This tool is a substantial improvement over the conventional approaches used to maintain archives. Of particular importance is the removal of library objects that are no longer needed. Many developers have spent hours debugging problems due to the presence of obsolete files remaining from older builds of the archives.

The ***qef*** system consists of about 80 tools, most of which have not been mentioned directly. Some of these tools are provided for use within construction recipes; however, many are useful in their own right.