Linux Standard Interface Validation Using CodeCheck

Porting C and C++ from any Operating System or Compiler to Linux

Linux Standard Base
Hardware Platform Interface
POSIX - SVID

CodeCheck™ is a product of Abraxas Software, Inc.

For more information, contact:
Abraxas Software, Inc.

Phone: 503-232-0540
Fax: 503-232-0543

Email: support@abxsoft.com
http://www.abraxas-software.com/

Table of Contents

Introduction - Why Programs Fail to Port

Purpose

The purpose of this document is to describe to how to use the Abraxas Software CodeCheck Source Code Analysis System as a tool that can help move C and C++ applications from general architecture's to LINUX architecture.

Abraxas CodeCheck was released in 1988 with the intended purpose of providing support for 'Virtual Compilation". That is the ability to behave as any C or C++ compiler on the market so that a developer could produce an application for any platform with out access to the target platform.

With Linux Standard Base [LSB] the problem of determining whether an application will port, and determining what must be modified to achieve the port can be accomplished with CodeCheck on any operating system and for any C or C++ compiler source. This means that from any computer the analysis can be accomplished, and that no matter where the C or C++ came from the analysis can be determined. The results of the analysis can be generated in any from from XML to XLS, or HTML to simple TexT. In fact CodeCheck can generate data in any forma you can imagine because it's fully and completely programmable.

Besides being able to analyze any C or C++ from any compiler vendor for any operating system CodeCheck can detect whether the source also meets any standard. Most common standards have already implemented by CodeCheck [SVID, POSIX, LSB, …]. Therefore for instance its possible from Microsoft Windows 2000 to analyze MSDEV C++ and determine if it will port to LINUX POWER 64 and meet the LSB requirements and determine if the source is POSIX [Linux LSB] compliant and determine any 64 bit porting problems. Therefore the source can be written portable the first time and customer wanting to port from Windows 2000 can determine the complexity of the task from the source origin.

Linux Specifications cover important areas of a Linux-based system's, including hardware support, and compatibility with other specifications - such as the Linux Standard Base (LSB), POSIX, Service Availability Forum (SA-Forum), Hardware Platform Interface (HPI).

Linux Standard Base Specification

The Linux Standard Base (LSB) defines a system interface for compiled applications and a minimal environment for support of installation scripts. Its purpose is to enable a uniform industry standard environment for high-volume applications conforming to the LSB.

The LSB defines a binary interface for application programs that are compiled and packaged for LSB-conforming implementations on many different hardware architectures. Since a binary specification must include information specific to the computer processor architecture for which it is intended, it is not possible for a single document to specify the interface for all possible LSB-conforming implementations. Therefore, the LSB is a family of specifications, rather than a single one.

The purpose of this document is to completely explain how the CodeCheck can be used to validate LSB compliant C and C++ from any source. More than just validation of course is finding the portation problems and fixing them. When a developer regularly uses CodeCheck to validate and test source portable LSB compliant source is always being generated.

Standards for the C and C++ Languages

There seem to be no fewer than six “standards” for the C and C++ language’s, all of which are covered by CodeCheck. Figure 1 depicts the family tree for C and C++ standards, with the earliest version on top:

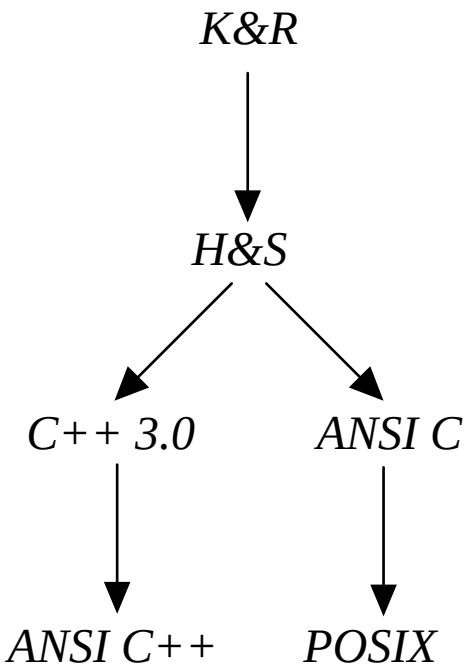

Figure 1: The Evolution of C and C++ Standards.

Each descendent of the original C has added significant extensions to the original language, while trying to remain true to the spirit of C.

- The **K&R** standard, as described in the first edition of Kernighan & Ritchie (1978). This is certainly the single most influential book in the history of C. The language was only loosely defined in this "standard," however, and it lacks many of the popular features that are commonplace now (*e.g.* enumerated constants, prototypes, the void type). Although obsolete, there are still many K&R compilers in daily use around the world.
- The **H&S** standard, as described in the first edition of Harbison & Steele (1984). This was the first careful description of the K&R standard, with many modern extensions included (*e.g.* the enum and void types). The H&S standard represents a transitional phase between K&R and ANSI. Most pre-ANSI compilers in use today are best described as adhering to the H&S standard.
- The **ANSI C** standard, as defined by the American National Standards Institute and certified internationally as ISO/IEC 9899. This version represented a significant advance in precision over H&S. It also introduced several significant innovations (*e.g.* the preprocessor paste operator).
- The **POSIX** standard, as defined by the American National Standards Institute and certified internationally as ISO/IEC 9945. Part 1 of this standard includes and extends the ANSI C standard, and details the interface and behavior of a standard library of operating system services.
- The **C++ 2.0** standard, as defined in "The Annotated C++ Reference Manual," by Ellis and Stroustrup (1990). This book is the base document for an ANSI committee that is now developing an official standard for C++.
- The **C++ 3.0** standard, as defined in "The C++ Programming Language Manual, **3rd Edition**" by Bjarne Stroustrup (1997). This book is the base document of the current ANSI C++ standard.

What is CodeCheck actually doing?

So far there has not been a lot of discussion about Virtual-Compilation. When we refer to compilation we're not referring to code generation. We referring to syntax analysis and data-base generation, the so called front-end of a C or C++ compiler.

The secret behind CodeCheck is the rule-files, they're a C-like script that tells the CodeCheck expert-system what to do with source code after it has built a database.

A CodeCheck rule-file program looks just like a very simple C program. Indeed, CodeCheck programs are written using a small subset of the C grammar, so anyone who can read C can also read CodeCheck. A CodeCheck program is, in fact, just a collection of if-statements (called "rules") and variable declarations. The

CodeCheck interpreter translates this collection of rules into pseudo-code, which is used during the analysis of a C source to control the code checking operation.

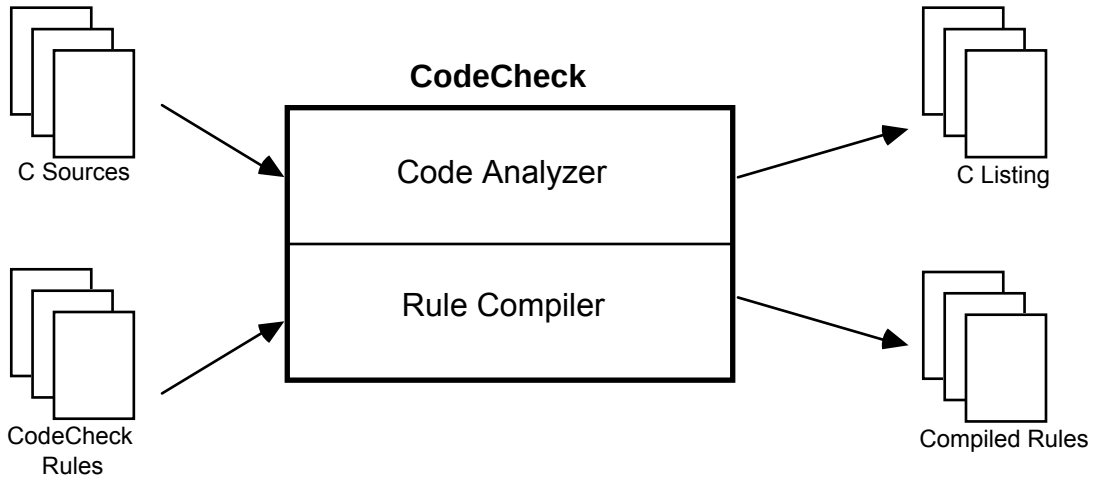

Figure 2: Actions of the two components of CodeCheck.

To analyze a C source file, the user has only to specify the name of the C source file and the name of the CodeCheck program. The CodeCheck program will be compiled (if necessary), and then the C source file is analyzed in accordance with the CodeCheck rules. As depicted in Figure 2, CodeCheck has two logically separate components — the Code Analyzer and the Rule Compiler.

Without rule-files CodeCheck behaves just like a typical compiler. If tells you whether the source is syntactically correct or not. With a rule-file codecheck can tell you anything about the source. Given that Abraxas Software has been writing rule-files for sixteen years we honest believe that we now have rule-files for most algorithms that people might want to apply to their source. This list of course includes the ability to ask questions like?? Is this code POSIX compliant?? Is it SVID compliant?? Does it embrace the Scott Meyer's Effective C++ writing development standards??

In summary CodeCheck is really two compilers and an expert system

1.) CodeCheck compiler "Code Analyzer" processes your input source code for emulation and builds a database.
2.) CodeCheck compiler "Rule Compiler" pre-compiles the rule-file and builds a expert system tree. Rule-Files always have the extension suffix .cc [dot-cc]
3.) During actual processing CodeCheck applies the rule-file expert system tree using reverse and forward chaining to the source code database. This results in the maximum amount of knowledge being extracted from any source code you wish to analyze. A "rule" in a rule-file is called an "event deferred trigger". It's important to know when studying codecheck rule-files that they are NOT procedural.

Of course don't let the above scare you about performance all the three steps are done in a single pass. Since most common algorithms have already been implemented most people who use CodeCheck never have to actually learn how to write rule-files. All source for ALL rule-files are included with CodeCheck to aid in learning how to solve generic problems involving portability analysis.

Using CodeCheck without a rule-file is a lot like a "virtual-lint". When a rule-file is applied virtually any analysis is possible. We like to differentiate two kinds of analysis Subjective and Objective. Objective would be like find all the 64 bit problems in a source file. A subjective analysis would be report back if the code is commented in "Old English". We can and will detect ALL objective missions. We generally cannot automate subjective missions, because generally no two people can agree on an outcome.

You can download a short list of rule-files at http://www.abxsoft.com/rule_idx.txt

To take a close look at 100's of real-world solutions to problems download http://www.abxsoft.com/dl/ccrules.zip

Chapter 1 Virtual Compilation

1.1 What is Virtual Compilation

CodeCheck was originally designed for the mission of “virtual compilation” in 1986. A major problem that Abraxas Software had was that the predominate market was the IBM-PC for product sales, but most of our developers wanted to work at home with Mac’s or Next Machines. There was a constant frustration that the software wouldn’t compile on target Windows OS compilers as some of our BEST programmers wouldn’t have an PC in their home! The solution - We developed CodeCheck a general purpose programmable tool that could determine for instance that on a Macintosh with the programmer using the MPW C++ compiler a means where the programmer could easily ‘emulate’ a IBM-PC C or C++ compiler. This simple test validated that all code sent to Abraxas Software would compile with the PC C/C++ compilers of the day.

In 1988 we realized that the “drop-in” rule-files we developed for virtual-compilation could also be used to enforce company programming standards and metrics. We released CodeCheck to the common market. While most of our users didn’t seemed to care much about Virtual-Compilation that feature was forgotten.

Today in 2004 we see the same problem as 1986, everyone is using different versions of Linux, and using different machines. Our USA customers for instance may have intel-386 running Red-Hat in their home, while our European customers run Debian. Quite often when a developer ship’s so called ‘portable-source-code’ applications to their customer the target compiler may not properly compile the application or worse there may be a hidden bug that the compiler is incapable of detecting. Our mission is to deliver tools to detect migration problems, and to assist in developing platform independent C and C++ for Linux of any variant.

1.2 How does Virtual Compilation work??

At the time Abraxas Software invented CodeCheck C++ had already blasted off in 1984, and the first C++ conference had taken place in 1986. The predominate C compilers of the day were VAX, IBM-PC, … and of course UNIX.

C++ was a research curiosity. Supporting dozens of C compilers was essential. Today C++ is the common denominator of new development, and C is largely a maintenance issue for developers having to maintain legacy systems. Then again there many who still use C as the backbone of their application development, as there is nothing more portable than classic standard C using the standard-portable-library. Which leads the story to Linux Base Standard.

Below is a console dump from Windows 2000 showing some of the codecheck command option switches.

```
F:\Tmp\LSB>chknt
Abraxas Software (R) CodeCheck NT version 10.04 B5
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.

COMPILER KEYWORD OPTIONS: (default is ANSI C with extensions)
-K0 Strict 1978 K&R C         -K4 Standard  C++    -K8  GNU C/C++ IBM-VA C++
-K1 Strict ANSI C             -K5 Symantec  C++    -K9  Metrowerks CW C++
-K2 K&R C with extensions     -K6 Borland   C++    -K10 VAX & HP/Apollo C
-K3 ANSI C with extensions    -K7 Microsoft C++    -K11 Metaware High C
```

As can be seen above the switch –Kn selects the emulation desired. For instance –k8 selects IBM-VACPP mode. –k1 strict ansi-c, and –k4 for classic ansi-c++.

Note that we support ALL C compilers back to the early 1970's vintage K&R C, and to the latest ANSI-C++ standard document. It's quite straight forward with codecheck to analyze twenty year old HP/Apollo C and determine whether the source is portable to POWER-LINUX, and if not what changes will be required.

The –Kn switch actually puts code-check into the mode required and selects the appropriate internal compiler, and some intrinsic macros. The CCP add's any additional flavor-codes that are required for a particular emulation. Given the dozens of Linux permutations of versions and vendors one can appreciate that there are many CCP's files for Linux emulation.

To analyze MSDEV C++ from win2k and see how the code would port to GNU-GCC one would type. Please remember that the command-line examples are only a means to discuss invocation most users activate analysis by pull-down menus via their favorite GUI source-code-editor.

```
check rh386_cpp.ccp test.cpp
```

Where rh386_cpp.ccp is the configuration file to tell codecheck to operate in gcc mode using the appropriate red-hat system include headers. The rh386_cpp.ccp file is the 'brains' this tells codecheck what it needs to know to process the red-hat include files for 2.91 c++ on a standard red-hat linux intel distribution.

Rh386_cpp.cc:

```
# codecheck for red-hat intel386 c++ compiler
# standard definition "gcc -v test.cpp 2> test.c.txt"
-k8                      // -k8 puts codecheck in GCC-GNU MODE
# define's               // i386 redhat gnu needs these macros
-D__GNUC__=2
-D__GNUG__=2
-D__cplusplus
-D__GNUC_MINOR__=91
-D__ELF__
-Dunix
-Di386
-Dlinux
-D__ELF__
-D__unix__
```

```
-D__i386__
-D__linux__
-D__unix
-D__i386
-D__linux
-D__EXCEPTIONS
-Di386
-D__i386
-D__i386__
-D__tune_i386__
#inlude paths           // header file search list - order critical
-I/usr/include/g++-2
-I/usr/i386-redhat-linux/include
-I/usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include
```

For standard compiles like msdev c++ the codecheck cmd line for emulation is only.

check -k7 test.cpp

Also c++ checking using the solaris compiler is just

Check -k4 test.cpp

Which is quite simple this is because things in the MS world are quite standard. Linux on the other hand is quite complex like the above shows. The good news is that Abraxas Software already has all the config files [CCP] for most linux compilers. We like to compare gnu-gcc.

CodeCheck can be used to 'emulate' any C and/or C++ compiler on any operating system, because we at Abraxas Software support ALL operating systems. All versions of CodeCheck supports ALL compilers. This means that from the Red-Hat i386 version of Linux you can run codecheck is do a target analysis to any platform you wish!

1.3 Examples of Virtual Compilation

Now were ready to show how to analyze particular C and C++ source files.

1.3.1 IBM VACPP

IBM VACPP is extremely popular on IBM AIX operating systems and also available on the Windows line of Microsoft operating systems. We don't have the Windows version of VACPP installed on this test system but we do have the AIX-VACPP compiler headers present. So here we're going to do a "virtual-compilation". A little c++ test file called test.cpp. This little snippet of C++ is out of a IBM Migration Guide. We'll be using this example code since it's a standard test case.

cat test.cpp

```
#include <iostream.h>            // using AIX VACPP IOSTREAM.H
class test{
public:
```

```
void test1( char const & nEvt )
 {
   cout << nEvt << endl;
 }
};
int main()
{
test t;
char * const ptr="ABC";
t.test1( *ptr );
return 0;
}
```

Now we'll virtually compile the file test.cpp. The CCP file aix.ccp is a configure file for Windows-2000 that configures CodeCheck to process AIX 5.0 IBM-VACPP. Note that also on this system in is the appropriate IBM-VACPP 5 header files. The –p switch tells codecheck to produce a verbose progress report.

Here is the CCP file that tells CodeCheck how to emulate IBM-VACPP. Note the '#' [pound-sign] in column one is a comment line.

C:\usr>cat aix.ccp

```
# -k8 to support GNU C++ and/or IBM VA
-k8
# define macros for AIX EMULATION
-D_AIX
-D_AIX50
-D_IBMR2
-D_POWER
#-d__IBMCPP__=500
#-d__TOS_AIX__
-D__ia64
# header Path's required. [ w2k testing env ]
-I\usr\include
-I\usr\include\sys
-i..
-I\usr\vacpp\include
```

The secret of course behind the emulation [virtual-compilation] is the configuration file aix.ccp as shown above it has three parts first the indication of –k8 which puts codecheck gnu/gcc [vacpp] mode, then the macros that the system headers need defined to know the target of emulation, and lastly the path's where codecheck can find the correct IBM-AIX-VACPP 5.0 header files. Header files are NOT always required, but a true correct emulation, what we call "Perfect Virtual Emulation" requires the correct header files. For information on CodeCheck without the use of header-files contact Abraxas Software *info@abxsoft.com*

C:\usr>chknt aix.ccp test.cpp -p

```
Abraxas Software (R) CodeCheck NT version 10.04 B5
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.

# Header files are in these directories:
     \usr\include\
     \usr\include\sys\
     \usr\vacpp\include\
test::test1
main
File test.cpp check complete.
Checking IBM-VA/GNU-GCC C++ file t.cpp with no rules:
```

Your probably not impressed but what we just did is process 100's of IBM-AIX VACPP header files and validate that all the code was correct. In actuality we haven't even started to use codecheck as there is no rule-file associated with this example. We're going to delay the actual usage of rule-files until the standards section where we're going to show how to apply POSIX [Linux LSB], and SVID standards to our test suite.

1.3.2 GNU/GCC

Like the Previous case now were going to virtually compile the test.cpp case from the IBM-VACPP case with the GNU-GCC headers and use a GCC codecheck CCP configuration file.

In this example a Unix Version of CodeCheck is running on Windows 2K and we're processing our test.c case using the configuration and headers from gcc-gnu 2.91.

First of all the CCP – CodeCheck Configuration Project file used to emulate the gnu-gcc 2.91 c compiler. Note accept for the commented out include path's everything is the same no matter which Operating System the analysis is done from. If we wished to NOT use header-files then a dot-eye could have been used. [See appendix for dot-eye]

cat rh386ansic.ccp

```
# codecheck ansi C config for redhat gcc default i386 2.91
# "gcc -ansi -x c -v test.c > test.c.txt"
-k8
#define
-D__GNUC__=2
-D__GNUC_MINOR__=91
-D__STRICT_ANSI__
-D__ELF__
-D__i386__
-D__linux__
-D__unix
-D__i386
-D__linux
-D__tune_i386__
#include from default gcc-gnu linux
# -I/usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include
# -I/usr/include
#include for win2k run above
-IF:\GCC\egcs-2.91.66\include
-IF:\GCC\usr-include
```

Our test.c is the simple "hello world" test case.

```
cat test.c
#include <stdio.h>
```

```
void main()
{
printf("Hello World");
}
```

Finally we apply CodeCheck to the simple C application and tell it to compile the code with a –P verbose progress report.

chkunx rh386ansic.ccp test.c -p

```
Abraxas Software (R) CodeCheck Unix version 11.01 B1
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.
# Header files are in these directories:
     F:\GCC\egcs-2.91.66\include/
     F:\GCC\usr-include/
Checking IBM-VA/GNU-GCC C++ file test.c with no rules:
#  Reading header file <stdio.h>  in  F:\GCC\usr-include/
#  Reading header file <features.h>  in  F:\GCC\usr-include/
#  Reading header file <cdefs.h>  in  F:\GCC\usr-include/sys/
#    Returning to file <features.h>
#  Reading header file <stubs.h>  in  F:\GCC\usr-include/gnu/
#    Returning to file <features.h>
#    Returning to file <stdio.h>
#  Reading header file <stddef.h>  in  F:\GCC\egcs-2.91.66\include/
#    Returning to file <stdio.h>
#  Reading header file <stdarg.h>  in  F:\GCC\egcs-2.91.66\include/
#    Returning to file <stdio.h>
#  Reading header file <types.h>  in  F:\GCC\usr-include/bits/
#  Reading header file <features.h>  in  F:\GCC\usr-include/
#    Returning to file <types.h>
#  Reading header file <stddef.h>  in  F:\GCC\egcs-2.91.66\include/
#    Returning to file <types.h>
#    Returning to file <stdio.h>
#  Reading header file <libio.h>  in  F:\GCC\usr-include/
#  Reading header file <_G_config.h>  in  F:\GCC\usr-include/
#  Reading header file <types.h>  in  F:\GCC\usr-include/bits/
#    Returning to file <_G_config.h>
#  Reading header file <stddef.h>  in  F:\GCC\egcs-2.91.66\include/
#    Returning to file <_G_config.h>
#    Returning to file <libio.h>
#  Reading header file <stdarg.h>  in  F:\GCC\egcs-2.91.66\include/
#    Returning to file <libio.h>
#    Returning to file <stdio.h>
#  Reading header file <stdio_lim.h>  in  F:\GCC\usr-include/bits/
#    Returning to file <stdio.h>
#    Returning to file "test.c"
main
File test.c check complete.
```

There are no errors, and normally we wouldn't have used the –P, had this been the test.cpp case the output would have been to long to show the include-path processing.

1.3.3 Microsoft Visual Studio C++

We're including the MSDEV C case for reference. There are lots of users that do intend to port Microsoft C & C++ to Linux, so we're going to show that case.

First of all in order to configure the environment so that the Microsoft Compiler can run, you must use their built-in configuration batch file “vcvars32”, which is created for your system at the time the MSDEV compiler is installed. This batch-file is placed in the /vc98/bin folder normally.

```
F:\GCC>vcvars32
Setting environment for using Microsoft Visual C++ tools.
```

Next it’s a good idea to verify that your compiler is correctly installed.

```
F:\GCC>cl -Zs test.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for
80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.
test.c
```

Given that the above –Zs “syntax check” option generated no error’s we’re now confident that the environment is correct. In order to have codecheck emulate the above compile its just …

```
F:\GCC>chknt -k7 test.c -p
Abraxas Software (R) CodeCheck NT version 10.04 B5
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.
# Header files are in these directories:
     C:\PROGRA~1\MICROS~2\VC98\ATL\INCLUDE\
     C:\PROGRA~1\MICROS~2\VC98\INCLUDE\
     C:\PROGRA~1\MICROS~2\VC98\MFC\INCLUDE\
Checking Microsoft C++ file test.c with no rules:
#  Reading header file <stdio.h>  in  C:\PROGRA~1\MICROS~2\VC98\INCLUDE\
#    Returning to file "test.c"
main
File test.c check complete.
```

The above may or may not be impressive, but the point to be explained later is that given the fact we can virtually emulate any MSDEV C or C++ project then we can apply our LSB or POSIX rule-files to this code, and find where the MSDEV code is not compliant. Then we can switch over to –k8 with the same MSDEV code and apply the gnu-gcc headers and determine how to make the codechek lsb compliant.

1.3.4 TRU64 C++

1.3.4 Sun Solaris C and C++

Chapter 2 API - Application Program Interface's

Application Program Interfaces are basically a standard way of calling subroutine functions using standards arguments and calling conventions. Ever since the days of K&R C back in the 1970's there were a small collection of library-routines called the C Runtime Library. Probably the most portable code ever written is code that is written in pure K&R C and that call's no other external function other than the basic standard C runtime. Things have gotten quite complex since the 1970's. Since 1982 we have C++, which took its own path, and then there was ANSI-C. If we consider Microsoft C++ and Borland C++, and later versions of Metrowerks C++ and dozens of other vendors its safe to say that we now have a C & C++ tower-of-babel.

Today there are literally dozens of C and C++ 'standards' and likewise dozens of version of Linux all calling themselves standard. Linux Standard Base is an attempt to create a standard means of requesting operating services through a standard call interface and argument type-list that is truly standard for all Linux systems. This is accomplished by including a standard header in all applications. In C this header is called "unistd.h". In C++ we include "unistd.h" and "cxxabi". When a developer includes these headers in his source code they effectively must use the functions per the interface or the compiler will not compile their application.

The purpose of this chapter is to provide an overview of how these interfaces actually work for C and C++ for targeting the Linux environment. The goal of writing software once, and having it running anywhere is an attainable goal and certainly these interfaces are a means of achieving that goal. Writing software that will not port not only destroys market opportunity for those wishing to migrate, but is a tremendous waste of human resource.

2.1 Interfaces

In this section the interfaces are discussed. Interface is basically the current marketing term for function-call for old-timers. The Interface's for C and C++ will be briefly discussed and then explicit difference between Gnu-Gcc and IBM-Vacpp will be given in the header section.

The important thing about these interfaces is that they define exactly what the C and C++ external function call's should look like. Calling functions outside of the scope of the project that do not match the names in the interface is considered a violation. If third-party libraries are used such as Rogue-Wave, ACE, Corba then those libraries must be verified that they do not call functions outside the scope of the interface.

2.1.1 The C Interface

The C interface is straightforward and well documented in the ABI specification - Linux Standard Base Specification 1.3. The C interface is defined by www.freestandards.org. There you can download lsbdev-base-1.2.2-1.i386.rpm. There models for most chips at that location. The base includes header files so you don't have anything other than CodeCheck to do the tests! However there is one file you do need and that is stdarg.h which is normally in /usr/include on Unix-Linux variants. What follows is first our test case test.c. Because this document was written with Microsoft-Word we're on Windows 2000 using MKS Unix command tools.

```
cat test.c
#include <stdio.h>            // stdio.h is coming from lsbdev-base
main()
{
printf("hello");
}
```

Now we're going to tell codecheck to process this code as ANSI-C using the lsbdev-base header files. First we'll show the CCP file required for emulation on windows 2000.

```
C:\linux-std\lsb>cat lsb_c.ccp
# strict ansi-c virtual compilation -k1
-k1
# required by lsb to set "size_t" - cpu target must be defined
-D__i386__=1
# search path - source -> lsbdev-base-1.2.2-1.i386.rpm
-IC:\linux-std\lsb\i386\base\include
# implicitly search /usr/include on unix for stddef.h
```

Now we'll apply codecheck to the problem. The -p is for a verbose progress report so we can see what is happening. As can be seen we are only using the LSB ABI headers.

```
C:\linux-std\lsb>chkunx lsb_c.ccp test.c -p
Abraxas Software (R) CodeCheck Unix version 11.01 B1
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.
# Header files are in these directories:
     C:\linux-std\lsb\i386\base\include/
     /usr/include/
# Rule files are in these directories:
     /usr/CodeCheck/rules/
     /usr/CodeCheck/
Checking ANSI C file test.c with no rules:
#  Reading header file <stdio.h>  in  C:\linux-std\lsb\i386\base\include/
#  Reading header file <types.h>  in  C:\linux-std\lsb\i386\base\include/sys/
#    Returning to file <stdio.h>
#  Reading header file <unistd.h>  in  C:\linux-std\lsb\i386\base\include/
#  Reading header file <types.h>  in  C:\linux-std\lsb\i386\base\include/sys/
#    Returning to file <unistd.h>
#  Reading header file <time.h>  in  C:\linux-std\lsb\i386\base\include/sys/
#  Reading header file <types.h>  in  C:\linux-std\lsb\i386\base\include/sys/
#    Returning to file <time.h>
#    Returning to file <unistd.h>
#  Reading header file <stddef.h>  in  C:\linux-std\lsb\i386\base\include/
#    Returning to file <unistd.h>
#    Returning to file <stdio.h>
#  Reading header file <wctype.h>  in  C:\linux-std\lsb\i386\base\include/
#  Reading header file <types.h>  in  C:\linux-std\lsb\i386\base\include/sys/
#    Returning to file <wctype.h>
#    Returning to file <stdio.h>
```

```
#  Reading header file <stddef.h>  in  C:\linux-std\lsb\i386\base\include/
#    Returning to file <stdio.h>
#  Reading header file <stdarg.h>  in  /usr/include/
#  Reading header file <standards.h>  in  /usr/include/
#    Returning to file <stdarg.h>
#  Reading header file <va_list.h>  in  /usr/include/
#    Returning to file <stdarg.h>
#    Returning to file <stdio.h>
#    Returning to file "test.c"
main
File test.c check complete.
```

In the above case what we have accomplished is a full Ansi-C analysis of our test case and have applied the LSBDev-Base API/ABI Library. Note at this point we didn't check the source or API in GNU-GCC mode, we told CodeCheck to behave as a strict-ansic compiler!

If we wanted to check the above case from the point of view of a gnu-gcc compiler with CodeCheck all we would have to do is over-ride the –k1 with a –k8

```
C:\linux-std\lsb>chkunx lsb_c.ccp test.c -k8
Abraxas Software (R) CodeCheck Unix version 11.01 B1
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.
Checking IBM-VA/GNU-GCC C++ file test.c with no rules:
File test.c check complete.
```

We didn't use –p this time, but as shown there no warnings. The results make sense because gnu-gcc is just a superset of ansi-c. If we tried to compile gnu-gcc keywords in ansi-c mode, then we would have problems.

2.1.2 The C++ Interface

The following is from the latest draft of the LSB spec. Note they explicitly do not support C++ …

"C++ Language: *Because of the immaturity of the C++ ABI (for name mangling, exception handling, and other such issues), we do not standardize any libraries for C++ in this version of the Linux Standard Base. In a future version of this specification, name-mangling rules will be specified so that C++ symbols can be mapped into symbol names in the object file.*
It seems to be possible, using existing Linux development tools, to write an application in C++ which complies with this rule by linking statically with libstdc++ and all other libraries containing C++. The following command illustrates how this may be accomplished:
g++ example.cc -Wl,-Bdynamic,-lc,-Bstatic"
*This leaves us with the option of NOT supporting C++ or attempting to emulate their suggestion, e.g. to emulate "****g++ example.cc -Wl, -Bdynamic, -lc, -Bstatic".***

The lsbdev-base/c++ header's #include<unistd> and #include<cstdlib> can be "Forced-Included" into source that is to be analyzed by codecheck when the header's are not explicit. This will verify that the only external interfaces used are those defined in libstdc++ and using the arguments defined. If #Includes are

explicit in the source then the source is self documenting, e.g. telling the reader that the code is portable.

Lastly, all the C interfaces can easily be applied to C++ with CodeCheck

2.2 Header Files

2.3 Base Libraries

The base library interfaces for libc include each of the following groups, each group represents header files that contain the data definitions. The general idea is that no user program should directly interface with the hardware. All user applications must interface with the following general system interface groups.4

1. RPC
2. System Calls
3. Standard I/O
4. Signal Handling.
5. Localization Functions
6. Socket Interface
7. Wide Characters
8. String Functions
9. IPC Functions
10. Regular Expressions
11. Character Type Functions
12. Time Manipulation
13. Terminal Interface Functions
14. System Database Interface
15. Language Support
16. Large File Support
17. Standard Library

Chapter 3 Standards – Applying Standard Interfaces

In this chapter the concept of CodeCheck rule-files is finally introduced. Here we explain exactly how CodeCheck be used for its actual design. The purpose of CheckCheck is to trigger on events in source code programs and act on those events. CodeCheck is fully programmable and any objective event that is programmable is possible to detect.

3.1 LSB

In this section we implement the LSB checking per lsbcc specification. Basically it's quite simple. The lsbdev-base has a collection of header files that replace the standard gnu-gcc header files. When a program is written and there is an external function call that doesn't meet the criteria of the ABI interface description in the header file "#include<unistd.h>" then an LSB violation is signaled. We have also added another LSB violation in this example that of detecting leading underscores which is also a violation of the specification.

3.1.1 LSB C Case

There are two little files hello.c and hello_bad.c. The first file has no problems. The second case has a call to a non-lsb library function _getpid(), furthermore the leading underscores are not allowed.

```
num < hello.c
1: #include <stdio.h>
2: #include <unistd.h>
3:
4: void main()
5: {
6:   printf("hello world: %d\n", getpid() );
7: }
```

Here we apply codecheck using the lsb.cc rule-file. The rule-file lsb.cc will be explained after both hello.c and hello_bad.c are processed.

```
C:\linux\lsb>chkunx lsb_c.ccp hello.c -rlsb.cc
Abraxas Software (R) CodeCheck Unix version 11.01 B1
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.
Checking ANSI C file hello.c with rules from lsb.cc:
File hello.c check complete.
```

The case hello_bad.c. Here _getpid() is being called, but it was not defined in the unistd.h header file. This will cause codecheck to signal it as a function call that was not prototyped.

```
num < hello_bad.c
1: #include <stdio.h>
2: #include <unistd.h>
3: void main()
4: {
```

```
5:   printf("hello world: %d\n", _getpid() );
6: }
```

```
C:\linux\lsb>chkunx lsb_c.ccp hello_bad.c -rlsb.cc
Abraxas Software (R) CodeCheck Unix version 11.01 B1
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.
Checking ANSI C file hello_bad.c with rules from lsb.cc:
hello_bad.c(5): Warning W0002: Interface _getpid Must has leading underscore
hello_bad.c(5): Warning W0001: Undefined reference to _getpid
File hello_bad.c check complete.
```

This section is quite similar to the earlier sections on virtual compilation and emulation. The difference is that we have added a codecheck rule-file lsb.cc.

```
cat lsb.cc
// linux standard base rule-file
if ( idn_no_prototype ) warn( 1, "Undefined reference to %s", idn_name()
);
if ( idn_function ) {
    if ( idn_name()[0] == '_' )
      warn( 2, "Interface %s has leading underscore", idn_name() );
}
```

Note in the above file lsb.cc that there are two events. The first event tells codecheck to print a message when a function call is made without a prototype. The second event says that if a function call is made that a message should be emitted if it has a leading underscore. This example is exactly like those in the section virtual-compilation except we have added lsb.cc with these two little ‘rules’ we have emulated the lsb-cc checking tool. Note, that our tool support all dialects of C and C++ and support’s all operating systems and is platform independent. Of course the real power of codecheck is when we add rule-files that have dozens if not hundreds of rules, and that’s where the real power of codecheck really begins to show.

3.1.2 LSB C++ Case

3.2 POSIX

3.3 SVID

3.4 HIP

Chapter 4 Porting Issues – Applying Rule Files

Big vs. Little Endian Problems

The 32 to 64 Bit Issues

Compiler Errors

Standard C++

Truncation and Conversion

In this section we will discuss the class truncation cases. First we'll show the code.

```
C:\linux\lsb\test>num<trun.c
   1: extern long dosomething(int);
   2:
   3:  int main(int argc, char *argv[])
   4:  {
   5:  int i1, i2, i3;
   6:  long l1, l2, l3;
   7:
   8:  /* implicit truncation occurs in the next 3 statements */
   9:  i1 = l1;
  10:  i2 =i2 *l2;
  11:  i3 = dosomething(l3);
  12:
  13:  /* use explicit casting to obtain the intended narrowing*/
  14:  i1 = (int) l1;
  15:  i2 = (int) i2 * l2;
  16:  i3 = (int) dosomething((int) l3);
  17:  }
```

Fairly self documenting, let's let CodeCheck do the documentation for us using the 'conversion' rule-file conv.cc. Note below that all cases of truncation have been automatically detected by CodeCheck in this example lines 9-11 are implicit truncation, and lines 11&16 are explicit. The at_cnv.cc is a few pages so we haven't displayed it here in the manual the rule-file can be obtained from https://www.abxsoft.com/dl/autotest.zip.

```
C:\linux\lsb\test>chknt trun.c -rat_cnv.cc -c
Abraxas Software (R) CodeCheck NT version 11.01 B1
Copyright (c) 1988-2003 by Abraxas Software, Inc. All rights reserved.
Checking extended ANSI C file trun.c with rules from at_cnv.cc:
trun.c(9): Warning W1007: An integer or a float truncated implicitly
trun.c(10): Warning W1007: An integer or a float truncated implicitly
trun.c(11): Warning W1007: An integer or a float truncated implicitly
trun.c(11): Warning W1007: An integer or a float truncated implicitly
trun.c(15): Warning W1007: An integer or a float truncated implicitly
File trun.c check complete.
```

Threading Issues

Appendix I Links and References

The following is a short list of references, and web sites that should be reviewed.

Abraxas Software CodeCheck, LSB Rule-Files and Documentation
http://www.abxsoft.com/pdf
System V Application Binary Interface - DRAFT- 22 June 2000
http://www.caldera.com/developers/gabi/2000-07-17/contents.html
DWARF Debugging Information Format, Revision 2.0.0 (July 27, 1993)
File system Hierarchy Standard (FHS) 2.2 http://www.pathname.com/fhs/
IEEE Standard for Binary Floating-Point Arithmetic http://www.ieee.org/
System V Application Binary Interface, Edition 4.1

http://www.caldera.com/developers/devspecs/gabi41.pdf
ISO/IEC 9899: 1990, Programming Languages --C
ISO/IEC 9899: 1999, Programming Languages --
CISO/IEC 14882: 1998(E) Programming languages --C++
Linux Assigned Names And Numbers Authority http://www.lanana.org/
Large File http://www.UNIX-systems.org/version2/whatsnew/lfs20mar.html
LI18NUX 2000 Globalization Specification, Version 1.0 with Amendment 4
http://www.li18nux.org/docs/html/LI18NUX-2000-amd4.htm
Linux Standard Base http://www.linuxbase.org/spec/
OpenGL® Application Binary Interface for Linux

http://oss.sgi.com/projects/ogl-sample/ABI/
OSF-RFC 86.0 http://www.opengroup.org/tech/rfc/mirror-rfc/rfc86.0.txt
IEEE Std POSIX 1003.2-1992 (ISO/IEC 9945-2:1993)
http://www.ieee.org/
System V Application Binary Interface PowerPC Processor Supplement
http://www.esofta.com/pdfs/SVR4abippc.pdf
The PowerPC ™ Architecture: A Specification for a new family of RISC processors http://www.austin.ibm.com
The PowerPC Architecture Book I changes
http://www-1.ibm.com/servers/eserver/pseries/library/ppc_ch g1.html.

The PowerPC Architecture Book II changes http://www-1.
ibm.com/servers/eserver/pseries/library/ppc_chg2.html
The PowerPC Architecture Book III changes http://www-1.
ibm.com/servers/eserver/pseries/library/ppc_chg3.html
POSIX 1003.1c http://www.ieee.org/
RFC 1952: GZIP file format specification version 4.3
http://www.ietf.org/rfc/rfc1952.txt

RFC 2440: Open PGP Message Format

CAE Specification, May 1996, X/Open Curses,
Issue 4, Version 2 (ISBN: 1-85912-171-3, C610), plus Corrigendum U018
http://www.opengroup.org/publications/catalog/un.htm
CAE Specification, January 1997, System Interface Definitions (XBD),Issue 5 (ISBN: 1-85912-186-1, 605)
http://www.opengroup.org/publications/catalog/un.htm
CAE Specification, January 1997, Commands and Utilities (XCU), Issue 5 (ISBN: 1-85912-191-8, C604)
http://www.opengroup.org/publications/catalog/un.htm
CAE Specification, February 1997, Networking
Services (XNS), Issue 5(ISBN: 1-85912-165-9,C523)

http://www.opengroup.org/
CAE Specification, January 1997, System
Interfaces and Headers (XSH),Issue 5 (ISBN: 1-85912-181-0, C606)
http://www.opengroup.org/publications/catalog/un.htm
The Single UNIX® Specification(SUS) Version 1 (UNIX 95) System Interfaces & Headers
http://www.opengroup.org/publications/catalog/un.htm
The Single UNIX® Specification(SUS) Version 3
http://www.unix.org/version3/
System V Interface Definition, Issue 3 (ISBN0201566524)
System V Interface Definition, Fourth Edition
Double Buffer Extension Library http://www.x.org/

X Display Power Management Signaling (DPMS) Extension, Library Specification
http://www.x.org/
X Record Extension Library http://www.x.org/
Security Extension Specification, Version 7.1 http://www.x.org/.*Chapter 1. Introduction*
X Nonrectangular Window Shape Extension Library Version 1.0
http://www.x.org/
MIT-SHM--The MIT Shared Memory Extension http://www.x.org/
X Synchronization Extension Library http://www.x.org/
XTEST Extension Library http://www.x.org/

X11R6.4 X Inter-Client Exchange (ICE) Protocol http://www.x.org/
X11R6.4 X11 Input Extension Library http://www.x.org/
X11R6.4 Xlib - C library http://www.x.org/
X/Open Portability Guide, Issue 4 http://www.opengroup.org/
X11R6.4 X Session Management Library http://www.x.org/
X11R6.4 X Toolkit Intrinsic http://www.x.org/
zlib 1.1.3 Manual http://www.gzip.org/zlib/

IBM-Visual-Age compilers can be found at http://**ibm.co**m/software/awdtools/vacpp/.

http://gcc.gnu.org
http://linuxppc64.org
http://**ibm.co**m/developerworks/linux/

Appendix II Creating CodeCheck Configuration Files

The purpose of this section is to carefully describe exactly how CodeCheck CCP file(s) are created for gnu-gcc on Linux systems. The purpose of the CCP is to configure CodeCheck so that the Linux System Compiler Headers are processed exactly as if they were being read by gnu-gcc compiler.

All the below examples MUST be done from the standard Linux command-line-console interface in order to duplicate the results as shown. The results will of course be completely different for Debian, Suse, Red-Hat, Mandrake, … It is very important to realize that every system is completely different. While it is possible to write portable software that doesn't change the fact that 'linux' is not consistent.

Here are the two standard references we use for C & C++ configuration file generation. It's very important to configure your system at the beginning. If there is more than one compiler installed on your system configuration is extremely important in order to avoid mismatching system header-files.

In the OLD-DAYS of UNIX there was only one path called /usr/include. Everything was quite simple. Today nothing is simple nor standard.

[See below for exact source code definition of hello.c & hello.cpp. It is extremely important to do exactly as shown in our examples in order to obtain consistent and correct data.]

```
gcc -v hello.c > hello.c.txt        // pipe output to hello.c.txt
```

The output generated will appear as …

```
-D__GNUC__=2 -D__GNUC_MINOR__=91 -trigraphs -D__STRICT_ANSI__ -D__ELF__
-D__unix__ -D__i386__ -D__i386__ -D__linux__ -D__unix -D__i386 -D__linux
-D__i386 -D__i386__ -D__tune_i386__ hello.c
GNU CPP version egcs-2.91.66 19990314/Linux (egcs-1.1.2 release) (i386
Linux/ELF)
#include "..." search starts here:
 /usr/i386-redhat-linux/include
 /usr/lib/gcc-lib/i386-redhat-linux/egcs-2.91.66/include
 /usr/include
End of search list.
```

Note that by using –v gcc dumps all the required macros and include paths, and the proper order. The order is extremely important, the exact same order and path's as provided by "gcc –v" must be used for configuring CodeCheck for this particular emulation.

[*Your system may require "2>" rather than ">" to capture the stderr output*.]

```
gcc -v hello.cpp > hello.cpp.txt     // pipe output to hello.cpp.txt
```

From the above we can build you a custom gcc_c.ccp & gcc_cpp.ccp for your system.

** hello.c // MUST BE JUST LIKE THIS

```
#include <stdio.h>
main() { printf("hello"); }
```

** hello.cpp // MUST BE JUST LIKE THIS

```
#include <iostream.h>
main() {  cout<<"hello"; }
```

GCC on ALL operating system

Most users are running GCC on AIX, Sun-Sparc, Windows, and of course Linux. There are others, but I have just listed the most common. CodeCheck of course supports ALL GCC for ALL operating systems. Below we're only expanding the GCC on Windows case, the other cases are similar.

All information required to construct a CCP file is shown below. The information came directly from the **"cc -c -v hello.c"** information shown above. The –V verbose switch tells GCC to dump the exact configuration that it is using. We simply feed this information to CodeCheck in order to obtain a perfect emulation.

GCC C On Windows 2K

Given the "hello.c" case above compilation with codecheck on windows would be

check gnu_c.ccp hello.c

The contents of gnu_c.ccp are ...

```
::gnu_c.ccp // start of file

#force GNU GCC C compiler mode
-k8
-D__extension__=
-D__inline__=inline

# gcc C macros for 686 cygnus
-D__GNUC__=3
-D__GNUC_MINOR__=2
-D__GNUC_PATCHLEVEL__=0
-D__GXX_ABI_VERSION=102
-D_X86_=1
-D__NO_INLINE__
-D__STDC_HOSTED__=1
-Di386 -D__i386
-D__i386__
```

```
-D__tune_i686__
-D__tune_pentiumpro__
-D__tune_pentium2__
-D__tune_pentium3__
-D__i386__
-D__i386
-D__CYGWIN32__
-D__CYGWIN__
-Dunix
-D__unix__
-D__unix

# #include search paths for C cygnus 686
-I/usr/include/w32api/
-I/usr/lib/gcc-lib/i686-pc-cygwin/3.2/include/
-I/usr/include/

::GNU_C.CCP             // end of file
```

GCC C++ On Windows 2K

```
C++ is is very similiar to C above, basically the major change is the
path's for the header files.

Usage is ...

check gcc_cpp.ccp hello.cpp

::GCC_CPP.CCP     // START OF FILE

# config codecheck for GCC/GNU extensions

-k8

# cygnus 686 gcc c++ macros
-D__GNUC__=3
-D__GNUC_MINOR__=2
-D__GNUC_PATCHLEVEL__=0
-D__GXX_ABI_VERSION=102
-D_X86_=1
-D__NO_INLINE__
-D__STDC_HOSTED__=1
-Di386
-D__i386
-D__i386__
-D__tune_i686__
-D__tune_pentiumpro__
-D__tune_pentium2__
-D__tune_pentium3__
-D__i386__
-D__i386
-D__CYGWIN32__
-D__CYGWIN__
-Dunix
-D__unix__
-D__unix

# cygnus c++ header file path's
-I/usr/include/w32api
-I/usr/include/c++/3.2
-I/usr/include/c++/3.2/i686-pc-cygwin
-I/usr/include/c++/3.2/backward
-I/usr/lib/gcc-lib/i686-pc-cygwin/3.2/include
-I/usr/include
```

Appendix III - Dot-Eye File – Source Preprocessing

When checking new code, especially C++, don't use the .cc rule-files, e.g. don't add the -R, just make sure it compiles first. You should only use the –R after you have verified that the code compiles. This verification is to ensure the configuration is correct and the correct header-files are being used.

Always use command line as shown below, never use an GUI-IDE when testing & debugging problems. The time to use CodeCheck in the GUI mode is after you have determined that everything is working correctly. The great thing about GUI's is they hide the internals, but in the compiler business we desire to have nothing hidden.

There are three instances that you should consider using dot-eye files.

1.) You want to debug a source file and don't want to bother with configuration. The Dot-Eye method simply lets you create one file that has all the header-file and macro problems resolved making use with codecheck trivial. For instance you have a complex gnu-gcc file foo.cpp. To create the dot-eye its just "gcc –E foo.cpp > foo.i". Then from codecheck its just "check –k8 foo.i".

2.) You need to determine if your debug problem is in codecheck or the configuration. In general if codecheck can process the dot-eye file, it means that your missing a critical piece of information in your CCP file. Like the dot-eye method shown above if you your "check gcc.ccp foo.cpp" fails, but "check –k8 foo.i" works, then you have a problem in the ccp config file.

3.) Sometime for production testing its impossible to pass header-files to QA, quite frequently it makes more sense to deliver dot-eye files for checking. Thereby eliminating any configuration issue for QA. In this case you simply add the –E to the production makefile for the application, and pass the generated dot-eye files to QA, and have them process these rather than the source. CodeCheck rule-files [-R] can be configured to operate on dot-eye files and obtain almost as much information as the original source, and QA doesn't have to know anything about configuration. Using this method QA can treat ALL source as a black box.

Windows 2000 Compiler Verification

Always verify first at the command line that the correct "vcvars32.bat" [win2k] is ran first in order to set the appropriate header path's. Inspect 'set include" to verify that the correct headers are being used.

```
cl -Zs -TP yourfile.cpp
```

Verify MSDev C++ is correct. Check if codecheck can compile the code …

```
check -k7 yourfile.cpp
```

If this doesn't work then go back to 'cl -Zs', if you need -D's you'll need them here!

Now using msdev c++ compiler create a dot-eye file on windows 2000.

```
cl -E -TP yourfile.cpp > y.i
```

Pre-process the source and generate the dot-eye file.

Go back to msdev and try “cl -Zs -TP y.i” to verify that y.i compiles with msdev c++. Lastly use codecheck to process the dot-eye knowing the file is correct.

```
check -k7 y.i
```

GNU-GCC Compiler Verification

UNIX or LINUX only requires that the above "cl" become "cc" or "gcc", and that "-E" normally remains the same, see your "UNIX" 'man pages'

For GCC-GNU the “gcc –v” must be used to determine the correct header file search paths as there is no automatic way to generate the macro and include file path data. Its very important to realize that GNU pays attention to the file extension.

For IBM VACPP the generation of the dot-eye file is just “cc –P test.cpp” and test.i will be generated.

Appendix IV – Emulation – Virtual Compilation

CodeCheck can emulate any C/C++ compiler.

But in order to emulate codecheck must have the same information, e.g. macro's, includes, ... Everything CodeCheck does must be the same as the compiler you want to emulate. That means that CodeCheck must be able to read all the files that the original compiler reads. It means that CodeCheck needs to know all the macro's at the begin of compilation that are built-in to the compiler, and there exact values for the version of the compiler you intend to emulate.

If you have a problem the first thing to do is verify the code with the compiler [sparc, ... aix ...]. Using the compiler compile with the -V option [verify] and note location exactly of the correct header files. You'll need this later.

When the compiler runs it has built-in macro value constants. CodeCheck also must know the value of these constants exactly. They're dependent upon the compiler, version, and build. These constants make the header files make correct assertions at the #ifdef controls when when read reading header-files. If the correct values are not provided then codecheck cannot emulate the compiler. CodeCheck must process the #ifdef's in the headers exactly like the original compiler. So the '#if-test' values must be the same.

After you have verified with the compiler that your test case works, and you have done the above then try using codecheck with just the -k4 option when emulating Std C++. Make sure that the -E output of the compiler is exactly the same as the check.lst from codecheck, if it is different then you have a macro problem, and need to provide codecheck the correct macros.

When the check.lst looks just like the file-dot-eye [cc -E file.cpp > file.i], then you know you have the right macros and header files. A check.lst is generated by using codecheck on pure C or C++ using the –L –M and –H switches.

To determine the correct headers you must run the compiler in verify mode to learn the header paths chosen. To learn the macro values of all compiler intrinsic's you need to printf() them with a small program using your compiler, normally the printed values published by compiler vendors are obsolete in the man pages of the compiler. Very few manufacturers actually provide the current values generated by the compiler.

Migration Glossary

Compiler - The compiler [cc, cl, gcc] your trying to emulate.

Macro's - #define, or -D on cmd line [the compiler has these built into the cl.exe or gcc.exe ...]

Intrinsic macros - Macros that the compiler sets internally. These are macros that are built-in to the compiler.

GCC-GNU-G++ -The open source GNU Compiler Set, including gcc, the GNU C Compiler, g++, the GNU C++ compiler. GCC supports both true AT&T C++ [CodeCheck –k4], and GCC non-portable extended C++ [-k8]. For maximum portability always run GCC in Strict-Ansi mode [*gcc -ansi*].

Emulation – Using a CCP [codecheck project] file to force CodeCheck to process an source code application exactly as if the entire source file was being processed by the original target compiler. This allows CodeCheck to verify compatibility from any operating system for any C or C++ compiler on the market.

GPL -The GNU Public License, under which the Linux kernel and much of the software found in the open source community is licensed. More about the GPL can be found at http://www.gnu.org.

Header Files – Operating System and Compiler #include Files supplied by compiler maker & third parties that create interfaces to the compiler runtime library [stdio.h, iostream.h, vector ...]. System-header file's are loaded into /usr/include at system install and copied to the system when a compiler is installed. Lastly, headers may be imported from third party library's [ACE, CORBA, Rogue-Wave]. In a user application system headers must always be included before third party headers. Force-Include options can be used to force a system header to be processed first for non-conforming applications.

IA32 - The 32 bit Intel architecture. Generally the compiler header files used by Intel are quite similar to the Microsoft Visual Studio compiler headers.

Linux POWER - The Linux operating system running on IBM POWER hardware. It can be assumed in the context of this document that this refers explicitly to IBM iSeries and pSeries servers.

SuSE Linux - Enterprise Linux offering from SuSE Linux. In the context of this document, SLES is for the POWER architecture.

Visual Age Compiler – These tradition Open-Edition compiler's are functionally identical to the high-performance IBM compiler'st for AIX, and are available for Linux POWER. The VA compiler set includes xlC, the VA C compiler.

Visual Age C++ Compiler – The IBM VAC++ compiler for AIX and Power-Linux. From the point of view of CodeCheck IBM-VACPP is the same as GCC in –k8 mode, e.g. extended keyword.

Index