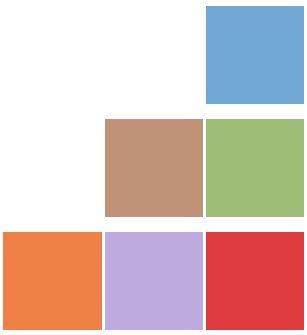
COBOL-IT® Compiler Suite Enterprise Edition

Getting Started
Version 3.11



Contents

Acknowledgment

This documentation is derived from COBOL-IT Source code, parts of which are derived from OpenCOBOL.

In 2008, COBOL-IT forked its own compiler branch, with the intention of developing a fully featured product and offering professional support to the COBOL user industry.

Third-Party software components embedded in the SOFTWARE and Services and submitted to specific licenses:

VBISAM

* Copyright (C) 2003 Trevor van Bremen

* Copyright (C) 2008-2018 COBOL-IT

* License: LGPL

GMP (GNU Multiprecision Library)

* Copyright 1991, 1996, 1999, 2000, 2007 Free Software Foundation, Inc.

* License: LGPL

GNU LIBICONV

The libiconv libraries and their header files are under LGPL.

product names are trademarks or registered trademarks of the holders of those trademarks.

Contact Information:

The Lawn
22-30 Old Bath Road
Newbury, Berkshire, RG14 1QN
United Kingdom
Tel: +44-0-1635-565-200

Compiler, Runtime, and Debugger Topics

Introduction

This document describes how to install and how to use the **COBOL-IT Compiler Suite.**

COBOL-IT License terms

COBOL-IT Compiler Suite

COBOL-IT Runtime System

COBOL-IT Corporate Headquarters are located at:
The Lawn
22-30 Old Bath Road
Newbury, Berkshire, RG14 1QN
United Kingdom
Tel: +44-0-1635-565-200

Installing COBOL-IT

The COBOL-IT compiler `cobc` requires that a “C” Compiler be installed on the host platform.

For a complete port list, including “C” Compilers that have been tested and are supported, see Port List with Supported “C” Compilers.

The COBOL-IT Compiler Suite Distribution

COBOL-IT Compiler Suite Unix/Linux distributions are provided in a gzipped tar file format. A

file-naming convention is observed when naming the distribution files. For example, the downloadable distribution for the Compiler Suite version 3.10.24 for the 64-bit Enterprise Linux x86 operating environments is named:

cobol-it-3.10.24-enterprise-64-x86_64-pc-linux-gnu.tar

COBOL-IT Compiler Suite Windows distributions are provided in a Windows Setup executable. The downloadable distribution for the Enterprise Edition of the Compiler Suite version 3.11 for Windows 64-bit operating environments is named:

cobol-it-3.10.24-enterprise-64-Windows-Setup.exe

Enterprise Editions of the COBOL-IT Compiler Suite are downloadable from COBOL-IT Online, with access provided by your Sales Representative.

Installing the binary distributions (Linux/Unix)

The COBOL-IT Linux/Unix 32-bit binary distributions are intended to be installed in **/opt/cobol-it.** The COBOL-IT Linux/Unix 64-bit binary distributions are intended to be installed in **/opt/cobol-it-64.** They are provided as **gzipped** tar files. As a root user, download the binaries that are compatible with your platform.

Consider the case for the file: cobol-it-3.10.24-enterprise-64-x86_64-pc-linux-gnu.tar.gz.

As Superuser, create the expected directory structure:

```
# cd /
# mkdir opt
```

Unpack the file:

```
# tar zxpf cobol-it-3.10.24-enterprise-64-x86_64-pc-linux-
gnu.tar.gz -C /opt
```

This will unpack the distribution in **/opt/cobol-it-64.**

Copy the license file to /opt/cobol-it-64.

```
# cp /home/cobolit/citlicense.xml /opt/cobol-it-64/citlicense.xml
```

Run the cobol-it-setup script to set all needed environment variables :

```
$ source /opt/cobol-it-64/bin/cobol-it-setup.sh
```

You are now ready to use the compiler!

Installing the binary distributions for the RuntimeOnly (Linux/Unix)

The COBOL-IT Linux/Unix 32-bit binary distributions for the RuntimeOnly are intended to be

installed in **/opt/cobol-it.** The COBOL-IT Linux/Unix 64-bit binary distributions for the Runtime are intended to be installed in **/opt/cobol-it-64.** They are provided as tar files. As a root user, download the binaries that are compatible with your platform.

Consider the case for the file: `cobol-it-3.10.24-enterprise-64-x86_64-pc-linux-gnu-runtimeonly.tar.gz`

As Superuser, create the expected directory structure:

```
# cd /
# mkdir opt
```

Unpack the file:

```
# tar zxpf cobol-it-3.9.21-enterprise-32-i786-pc-linux-gnu-runtimeonly.tar.gz -C /opt
```

This will unpack the distribution in **/opt/cobol-it-64.**

Copy the license file to /opt/cobol-it.

```
# cp /home/cobolit/citlicense.xml /opt/cobol-it-64/citlicense.xml
```

Run the cobol-it-setup script to set all needed environment variables :

```
$ source /opt/cobol-it/bin/cobol-it-setup.sh
```

You are now ready to use the runtime!

cobol-it-setup.sh

```
#setup the needed environement variables
DEFAULT_CITDIR=/opt/cobol-it-64
#
# Copyright (C) 2008-2009 Cobol-IT
#
if [ "x${COBOLITDIR:=}" = "x" ]
then
	if [ -f $DEFAULT_CITDIR/bin/cobol-it-setup.sh ]
	then
		COBOLITDIR=$DEFAULT_CITDIR
	else
		echo You must define  COBOLITDIR to the root instalation dir of COBOL-IT
	fi
fi
if [ "x${COBOLITDIR:=}" != "x" ]
then
	PATH=$COBOLITDIR/bin:${PATH}
	LD_LIBRARY_PATH="$COBOLITDIR/lib:${LD_LIBRARY_PATH:=}"
	DYLD_LIBRARY_PATH="$COBOLITDIR/lib:${DYLD_LIBRARY_PATH:=}"
	SHLIB_PATH="$COBOLITDIR/lib:${SHLIB_PATH:=}"
	LIBPATH="$COBOLITDIR/lib:${LIBPATH:=}"
```

```
    COB="COBOL-IT"
    SHLIB_PATH="$COBOLITDIR/lib:${SHLIB_PATH:=}"
    LIBPATH="$COBOLITDIR/lib:${LIBPATH:=}"
    COB="COBOL-IT"
    export COB COBOLITDIR LD_LIBRARY_PATH PATH DYLD_LIBRARY_PATH SHLIB_PATH
LIBPATH
    echo COBOL-IT Environement set to $COBOLITDIR
fi
```

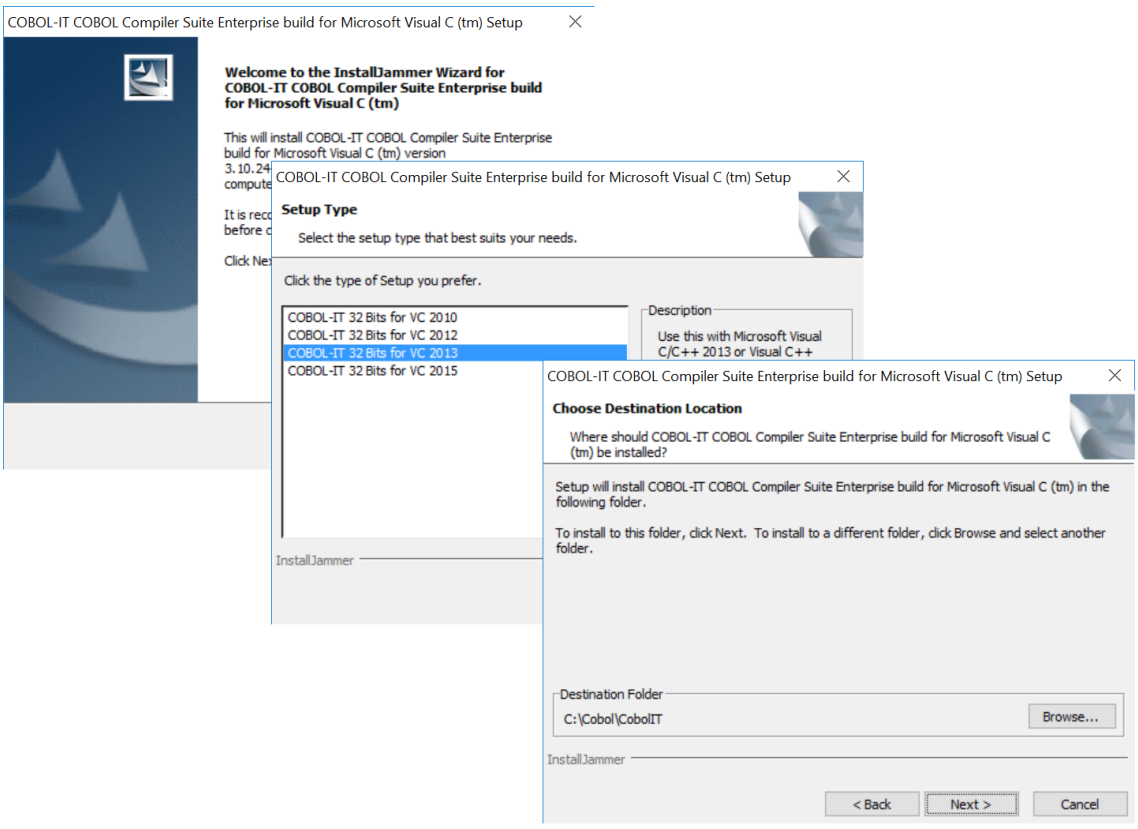
Installing the binary distributions (Windows)

The COBOL-IT Windows 32-bit binary distributions are intended to be installed in **C:\COBOL\COBOL-IT.** The COBOL-IT Windows 64-bit binary distributions are intended to be installed in **C:\COBOL\COBOLIT64.** They are provided as Windows Setup executable files.

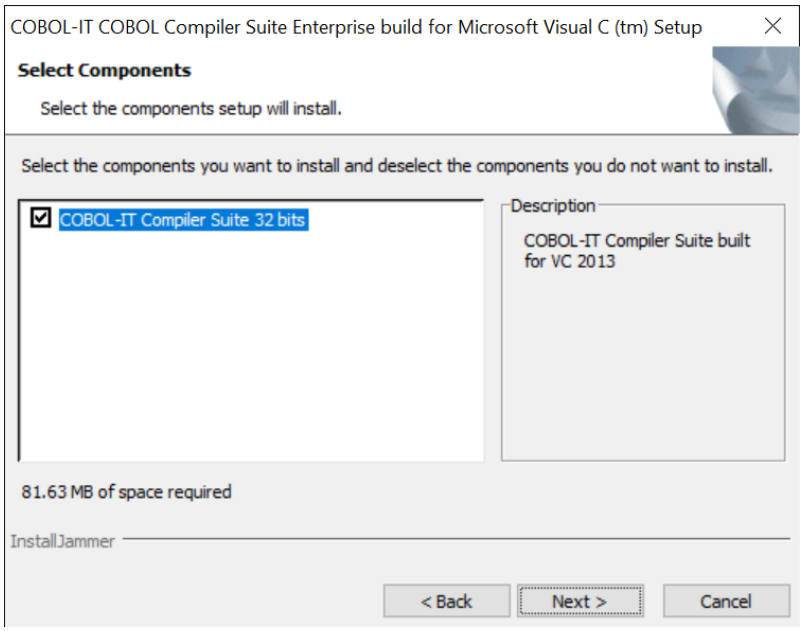
Click on the Windows-Setup executable to launch the setup.

Select the installation language, and click OK. To verify that you want to install the COBOL-IT COBOL Compiler Suite, click Yes.

Proceed through the initial install screen by clicking Next.
Select the "C" Compiler you have installed on your system.
Then, accept the default Destination Location by clicking Next.



Select the Component that you would like to install on your system, and click “Next”.
COBOL-IT Supports the 32- and 64-bit Visual C 2010, Visual C 2012, Visual C 2013 and Visual C 2015 Compilers.



After files are copied, click Finish.

The installation is complete!

This will create a Quick Launch Shortcut, which will cause the setenv_cobolit.bat file to be executed when you open your command shell. Setenv_cobolit.bat sets all the environment variables needed to begin using the COBOL-IT Compiler Suite.
You are now ready to use the compiler!

setenv_cobolit.bat (Windows 32)

COBOL-IT Supports the 32-bit Visual C 2010, Visual C 2012, Visual C 2013 and Visual C 2015 Compilers.

For the case described in the Installation above, where the user selected MS Visual C++ 2013, setenv_cobolit.bat checks for the installation of Visual C++ 2013, and exits if it does not find it.

```
@echo off
if "%VS120COMNTOOLS%_Z" NEQ "_Z" goto callVS12

echo COBOL-IT Compiler need MS Visual C++ 2013 to be installed.
pause
exit

:callvs12
call "%VS120COMNTOOLS%vsvars32.bat"
goto callcobit
```

```
:callcobit
if "%COBOLITDIR%_Z" NEQ "_Z" goto suite
set COBOLITDIR=%~dp0
echo SETTING COBOLITDIR=%~dp0

:suite
echo Setting Cobol-IT to %COBOLITDIR%
SET PATH=%COBOLITDIR%\BIN;%PATH%
```

setenv_cobolit.bat (Windows x64)

COBOL-IT Supports the 64-bit Visual C 2010, Visual C 2012, Visual C 2013 and Visual C 2015 Compilers. As an example, see below:

```
@echo off
if "%VS120COMNTOOLS%_Z" NEQ "_Z" goto callVS12

echo COBOL-IT Compiler need MS Visual C++ 2013 to be installed.
pause
exit

:callvs12
call "%VS120COMNTOOLS%..\..\VC\vcvarsall.bat" x64
goto callcobit

:callcobit
if "%COBOLITDIR%_Z" NEQ "_Z" goto suite
set COBOLITDIR=%~dp0
echo SETTING COBOLITDIR=%~dp0

:suite
echo Setting Cobol-IT to %COBOLITDIR%
SET PATH=%COBOLITDIR%\BIN;%PATH%
```

Installing a “C” compiler

If COBOL-IT does not detect the “C” compiler selected during installation on your Windows machine, it will return an error message. For the case where Visual C++ 2013 was selected during the installation process, the error message is constructed as follows:

```
echo COBOL-IT Compiler need MS Visual C++ 2013 to be installed.
```

For 32-bit Windows, the easiest way to get started is with a Microsoft® Visual Studio Community “C” compiler, which is a free download from the Microsoft website.

Supported versions are Visual Studio 2010, Visual Studio 2012, Visual Studio 2013, Visual Studio 2015.

Installing a "C" runtime

The Microsoft Visual C++ Redistributable Package installs the runtime components of Visual C++ libraries required to run applications developed with Visual C++ on a computer that does not have the same version of Visual C++ installed. As an example, COBOL-IT applications that are developed using Microsoft Visual C++ 2010 require that the Microsoft Visual C++ 2010 Redistributable package be installed in order to run.

This package is available to the user if they have Microsoft Visual C++ installed on their computer.

If they do not, they must download the appropriate Microsoft Visual C++ Redistributable Package from the Microsoft website.

Citlicense.xml

When a Subscription to the COBOL-IT Compiler Suite is registered, the registered user receives download authorization to the COBOL-IT Compiler Suite Enterprise Edition, and a license file called citlicense.xml, which is generated to match the duration of the registered subscription.
When a Subscription to the COBOL-IT Compiler Suite + RTS is registered, the registered user receives download authorizations to the COBOL-IT Compiler Suite Enterprise Edition, the Compiler Runtime, and a license file called citlicense.xml, which is generated to match the duration of the registered subscription, and enable the use of the Compiler and RuntimeOnly products.

Note that COBOL-IT Compiler Suite Enterprise Edition Subscriptions and licenses are defined as lasting for a prescribed period of time, from the date of the generation of the Subscription and corresponding license. That is, a one-year Subscription is accompanied by a one-year license, and the expiration date is set at one-year after the generation of the license, -not- one year after the installation of the software.

The COBOL-IT Compiler Suite Enterprise Edition (cobc), native executables created by the COBOL-IT Compiler Suite Enterprise Edition, COBOL-IT Runtime Enterprise Edition (libcobit.so, libcobit_dll.dll), and CitSORT search for a license file in the following manner:
Check to see if the COBOLIT_LICENSE environment variable is set. COBOLIT_LICENSE, if set, describes the full path, and license name to be used by the COBOL-IT Compiler, Compler-generated executables, and Runtime.

Example:
For Linux/Unix-based platforms:
>export COBOLIT_LICENSE=/opt/cobol-it/license/mycitlicense.xml

For Windows-based platforms:
>set COBOLIT_LICENSE=C:\COBOL\COBOLIT\license\mycitlicense.xml

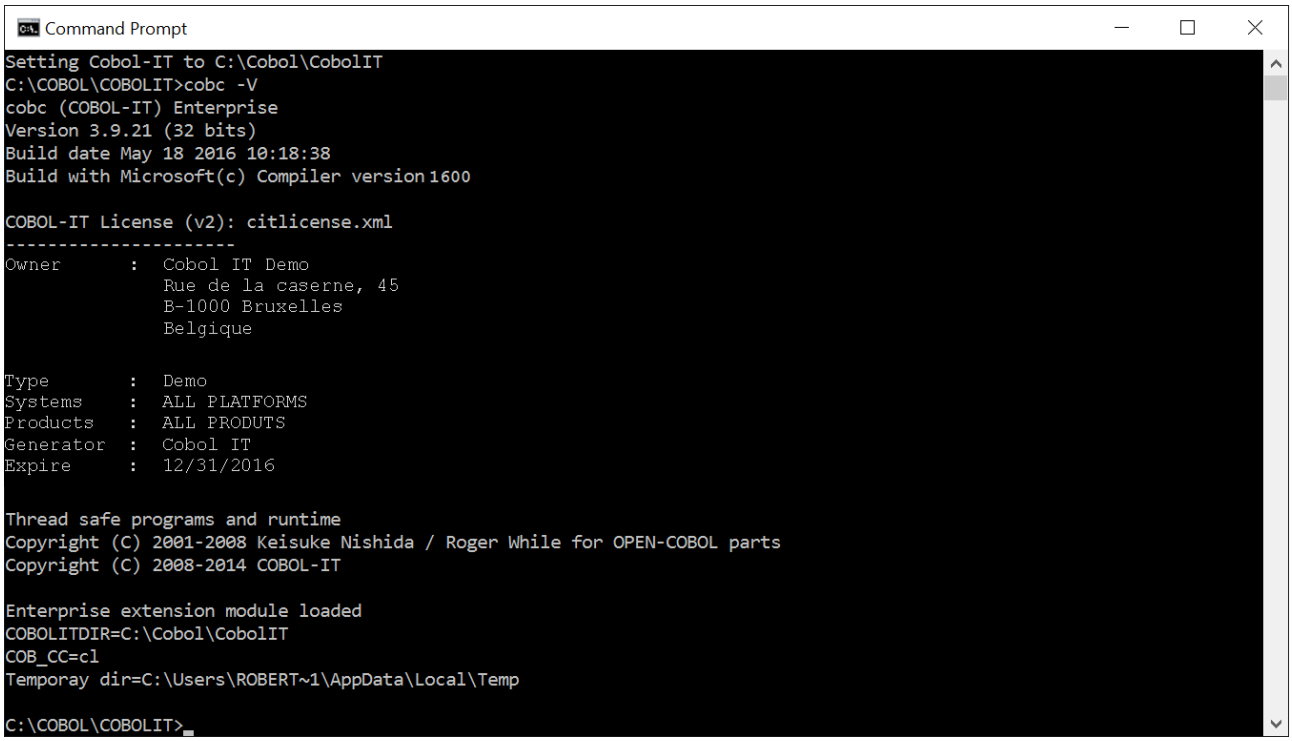
Check to see if the COBOLITDIR environment variable is set. If set, check for a file called citlicense.xml located in the directory defined by the COBOLITDIR environment variable.

If neither the COBOLIT_LICENSE or COBOLITDIR environment variables are set, check for a file called citlicense.xml located in the default installation directory. On Linux/Unix platforms, the

default installation directory is /opt/cobol-it for 32-bit product, and /opt/cobol-it-64 for 64-bit product. On Windows platforms, the default installation directory is C:\COBOL\COBOLIT for 32-bit product, and C:\COBOL\COBOLIT64 for 64-bit product.

For information about your Enterprise Edition license, type cobc –V

Information about the Location, Name, Owner, and Expiration Date of the license are shown below:



When a license expires, or is not found

If the COBOL-IT Compiler Suite Enterprise Edition (cobc), native executable created by the COBOL-IT Compiler Suite Enterprise Edition, COBOL-IT Runtime Enterprise Edition (cobcrun), or CitSORT fails to locate a valid license, they will exit, returning a message to the user as follows:

```
C:\COBOL\COBOLIT>cobc -V


Enterprise license is not validated.
Please Verify the environment variable COBOLIT_LICENSE value,
contact your local dealer or sales@cobol-it.com
```

Highlighting Compiler and Runtime Options

Source Format

COBOL-IT supports both fixed and free source format. The default format is the fixed format.

Example: Compile a free-format program.

```
>cobc –free hello.cbl
```

Example: Compile a fixed-format program

```
>cobc hello.cbl or
>cobc –fixed hello.cbl
```

Highlight

-free	Free format. In Free Format source code, the indicator area is in column 1. Area A starts in column 1, or immediately after an indicator. Area B starts in column 5. The beginning of the Identification Area is marked with a "|". The line ends when a carriage return/new line is detected.
-fixed	Fixed format. In Fixed Format source code, the indicator area is in column 7. Area A is in columns 8-11. Area B is in columns 12-72. The Identification Area is in columns 73-80. The line ends after 80 characters.

Shared object or native executable

COBOL-IT allows you to compile to a shared object format or to a native executable format. The default format is to compile to a shared object format. Running shared objects requires the COBOL-IT runtime cobcrun, whereas native executables can be run stand-alone.

Example: Compile to a native executable format, and run the executable.

```
>cobc –x hello.cbl
>hello
```

Example: Compile to a shared object format and run the executable.

```
>cobc –m hello.cbl
>cobcrun hello
```

Highlight

-m	Build a dynamically loadable module (default) -m compiles, assembles, and builds a dynamically loadable module/shared library. The output is saved in a .DLL file on Windows platforms, and in a .so file on Linux/Unix platforms.
-x	Build an executable program

-x builds a native executable. The output is saved in an .EXE file on Windows platforms, and in a file with no extension on Linux/Unix platforms.

Locating copy files

The COBOL-IT compiler can be directed to search for copy files in directories named by either the COBCPY, or COB_COPY_DIR environment variable, or with the use of the –I compiler flag. Copy file name resolution is refined with the use of the –ext compiler flag.

By default, the COBOL-IT compiler will search the current directory, and $COBOLITDIR\copy for named copy files, and if the copy files have no explicit file name extensions, the COBOL-IT compiler will search for default copy file extensions.

Default copy file extensions are:
– .CPY
– .COB
– .CBL
– .cpy
– .cob
– .cbl
– no extension

The COBOL-IT compiler will then check the environment variables COBCPY and COB_COPY_DIR for pathes to add to the default search pathes.

At the command line, you may add more directories to search with the –I <directory> compiler flag, and you may add more extensions to the default file extensions searched with the –ext <extension> compiler flag.

Highlight

-ext <extension>	Add file extension to list of default extensions
-I <directory>	Add <directory> to copy/include search path
COB_COPY_DIR	Path where standard copy books are stored by default: $COBOLITDIR/share/COBOL-it/copy in Unix/Linux, or $COBOLITDIR\copy in Windows.
COBCPY	List of directories to search for copy files Directories are separated by colon ":" on Unix machines, by a semi-colon ";" on Windows machines.

Example: Consider a case where a program, myprog.cbl has a copy file declared as follows:

```
COPY "customer.cpy".
```

And where "customer.cpy" is contained in a subdirectory called "copy".

```
>set COBCPY=.\copy
>cobc myprog.cbl
```

Example: Consider a case where a program, myprog.cbl has a copyfile declared as follows:

```
COPY "customer".
```

And where "customer.fd" is contained in a subdirectory called "copy". .fd is not a default extension, so both the directory and the extension need to be given to the compiler, to find the file.

```
>set COBCPY=.\copy
>cobc -ext=fd myprog.cbl    or

>cobc -I .\copy -ext=fd myprog.cbl
```

Redirecting Output to another Directory

When you compile your source code, you probably will want to create the compiled objects in a separate directory, and run them from this separate directory. To create compiled objects in a separate directory, use the –o compiler flag.

```
>cobc -o .\object hello.cbl
```

To run shared objects from a separate directory, set the COB_LIBRARY_PATH environment variable.

In Linux/Unix:

```
> export COB_LIBRARY_PATH=./object
```

In Windows:

```
 > set COB_LIBRARY_PATH=.\object
```

Execute your program:

```
>cobcrun hello
```

To run native executables from a separate directory, set the PATH environment variable.

In Linux/Unix:

```
> export PATH=./object:$PATH
>hello
```

In Windows:

```
>set PATH=.\object;%PATH%
>hello
```

Highlight

-o <dir>|<file> Compiler flag. Place the output into <dir> or <file>.

COB_LIBRARY_PATH Runtime environment variable. Directory where the shared objects that will be executed with cobcrun are located.

COBITOPT Compiler environment variable. Compiler flags to be used when running cobc, in addition to compiler flags named on the command line.

Note that compiler options can be stored in an environment variable called COBITOPT. The following are equivalent:

```
>cobc –o .\object hello.cbl and

>set COBITOPT=-o .\object
>cobc hello.cbl
```

Calling subprograms

When a COBOL program executes the statement CALL “myprog”, the COBOL-IT runtime performs the following step to resolve the symbol “myprog” :

* Search pre-loaded modules for the exact symbol. Modules will be pre-loaded by the runtime if they have been named by COB_PRE_LOAD runtime environment variable, and located by the runtime at startup.
* Search the running executable for that exact symbol. This search is successful if the call is to a “C” function that is statically linked to the runtime
* Search the current directory for the shared library myprog.cit. If found, the symbol is further searched for inside the share library. Note that when looking for a shared library, the name of the file searched for can be translated into upper, or lower case using the COB_LOAD_CASE runtime environment variable.
* Search the current directory for the shared library myprog.so in Linux/Unix, or myprog.dll in Windows. If found, the symbol is further searched for inside the share library. Note that when looking for a shared library, the name of the file searched for can be translated into upper, or lower case using the COB_LOAD_CASE runtime environment variable.
* Search the directory named by COB_LIBRARY_PATH for the shared library myprog.cit. As with the previously described search in the current directory, COB_LOAD_CASE may be used to translate into upper, or lower case.
* Search the directory named by COB_LIBRARY_PATH for the shared library myprog.so in Linux/Unix, or myprog.dll in Windows. As with the previously described search in the current directory, COB_LOAD_CASE may be used to translate into upper, or lower case.

* If a fully qualified shared library name is given, for example CALL “/mypath/MyFunc.so”, then the runtime searches for the requested file. If found, the runtime then looks for the following symbols :
  ○ base name without extension ('MyFunc') .
  ○ base name without extension, upper case ('MYFUNC') .
  ○ base name without extension, lower case ('myfunc') .

Highlight

COB_PRE_LOAD List of external modules pre-loaded at startup. Modules are separated by a colon ":" on Unix machines, by a semi-colon ";" on Windows machines.

COB_LOAD_CASE Runtime environment variable. When set to LOWER, the file name is converted to lower case. When set to UPPER, the file name is converted to upper case.

COB_LIBRARY_PATH Runtime environment variable. Directory where the shared objects that will be executed with cobcrun are located.

Using data files

COBOL-IT supports filename mapping, which allows a rich set of alternatives for aliasing file names in your COBOL programs. In the case where your program contains a statement such as

ASSIGN TO DATAFILE

You may locate the file in any of the following ways:

Set the environment variable DD_DATAFILE to [filename]
Set the environment variable dd_DATAFILE to [filename]
Set the environment variable DATAFILE to [filename]
Provide a [filename] of DATAFILE
Locate [filename] in a directory named by the COB_FILE_PATH environment variable
Highlight

Filename-mapping:yes Compiler Configuration file flag. Enables full set of file aliasing for resolving data file names.

DD_[filename] Runtime environment variable. [filename] is named in the ASSIGN TO [filename] clause. Set to the name of the file that the runtime will search for.

dd_[filename] Runtime environment variable. [filename] is named in the ASSIGN TO [filename] clause. Set to the name of the file that the runtime will search for.

[filename] Runtime environment variable. [filename] is named in the ASSIGN TO [filename] clause. Set to the name of the file that the runtime will search for.

COB_FILE_PATH Path to data files used by the application. COB_FILE_PATH is prepended to datafile names by the runtime as it works to resolve filenames and locations.

COB_FILE_TRACE When set to Y/YES, file tracing information is output to the file named by COB_ERROR_FILE, which includes information on how the runtime resolves file names on OPEN, and also status codes returned from unsuccessful file i-o operations.

COB_ERROR_FILE Specify the filename used to receive all runtime error messages that would otherwise be sent to stderr. When writing an error message, the runtime will create the specified filename if it does not exist, and will append to it if it does exist.

Options with multiple source files

COBOL-IT allows you to compile multiple source files into multiple shared objects, multiple source files into a single shared object, multiple source files into a single executable and to use a compiled native executable together with compiled shared objects. This wide range of capabilities gives you a full range of options for the deployment of your application in production.

Multiple source files to multiple shared objects

Example: Compile all programs with the –m option.

In Linux/Unix:

```
> cobc -m –o ./object main.cbl subr.cbl
```

This creates shared object files main.so and subr.so' in the ./object folder.

In Windows:

```
> cobc -m –o .\object main.cbl subr.cbl
```

This creates dynamic load libraries (DLL's) main.dll and subr.dll in the .\object folder.

Set the environment variable `COB_LIBRARY_PATH' to your library directory, and run the main program:

In Linux/Unix:

```
> export COB_LIBRARY_PATH=./object
```

In Windows:

```
> set COB_LIBRARY_PATH=.\object
```

Execute your program:

```
> cobcrun main
```

This causes the shared object main(.so/.dll) to begin executing. When subr(.so/.dll) is called as a subprogram from within main(.so/.dll), it will be located, as it is in the COB_LIBRARY_PATH, and it will be loaded and run.

Multiple source files to a single shared object

Example : Compile all programs with the –b option

In Linux/Unix:

```
> cobc -b –o ./object main.cbl subr.cbl
```

This creates a single shared object file main.so in the ./object folder.

In Windows:

```
> cobc -b –o .\object main.cbl subr.cbl
```

This creates a single shared object file main.dll in the .\object folder.

Multiple COBOL source files to a single executable

Example : Compile all COBOL source files with the –x option

In Linux/Unix:

```
> cobc -x –o ./object main.cbl subr.cbl
```

This creates a single executable file ‘main’ in the ./object folder.

In Windows:

```
> cobc -x –o .\object main.cbl subr.cbl
```

This creates a single executable file ‘main.exe’ in the .\object folder.

Using compiled executables with compiled shared objects

You may wish to create a main program as an executable, and have all CALL’ed subroutines be created as shared objects. You would not be required to store the native executable main program in the same directory as the shared objects. Note that the main native executable would be located by the PATH environment variable and the CALL’ed shared objects would be located by the COB_LIBRARY_PATH environment variable in this situation.

Example : Compile main program as an executable, subprograms as shared objects

In Linux/Unix:

```
> cobc -x main.cbl
> cobc –m –o ./object subr.cbl
```

This creates a single executable file ‘main’ in current working directory, and a single shared object, subr.so in the ./object folder..

In Windows:

```
> cobc -x main.cbl
> cobc –m –o ./object subr.cbl
```

This creates a single executable file ‘main.exe’ in the current working directory, and a single shared object, subr.dll in the ./object folder.

Set the environment variable `COB_LIBRARY_PATH' to your library directory, and run the main program:

In Linux/Unix:

```
> export COB_LIBRARY_PATH=./object
```

In Windows:

```
> set COB_LIBRARY_PATH=.\object
```

Execute your program:

```
> .\main
```

This causes the executable module main (or main.exe in Windows) to begin executing. When subr.so/subr.dll is called as a subprogram from within main, it will be located, as it is in the COB_LIBRARY_PATH, and it will be loaded and run. Note if main is not located in the current directory, then the directory where it is located must be referenced in the PATH environment variable.

COBOL source files and C source files to a single executable

Example : Compile a COBOL source files and a C source file with the –x option

In Linux/Unix:

```
> cobc -x –o ./object main.cbl subr.c
```

This creates a single executable file ‘main’ in the ./object folder.

In Windows:

```
> cobc -x –o .\object main.cbl subr.c
```

This creates a single executable file ‘main.exe’ in the .\object folder.

Separating the compile and link steps

COBOL-IT allows you to separate the compile and link steps. This is done using the –c compiler flag, which is equivalent to the “C” compiler –c flag. The program must be linked with the –x option. Note- This could be a useful thing to do, if you need to link a pre-compiled object with your COBOL.

Separate compile and link steps for multiple COBOL source files

Example: Separate the compile and link steps of several COBOL source files

In Linux/Unix:

```
> cobc -c subr1.cob      (produces subr1.o as output)
> cobc -c subr2.cob      (produces subr2.o as output)
> cobc -c main.cob            (produces main.o as output)
> cobc -x -o ./object main.o subr1.o subr2.o
```

This creates a single executable file 'main' in the ./object folder.

In Windows:

```
> cobc -c subr1.cob      (produces subr1.obj as output)
> cobc -c subr2.cob      (produces subr2.obj as output)
> cobc -c main.cob            (produces main.obj as output)
> cobc -x -o ./object main.obj subr1.obj subr2.obj
```

This creates a single executable file 'main.exe' in the ./object folder.

Linking "C" and COBOL objects

Example: Using objects created from "C" and COBOL together

In Linux/Unix:

```
> cc -c subrs.c          (produces subrs.o as output)
> cobc -c main.cob       (produces main.o as output)
> cobc -x -o prog main.o subrs.o
```

This creates a single executable file 'main' in the ./object folder.

In Windows:

```
> cl -c subrs.c          (produces subrs.obj as output)
> cobc -c main.cob       (produces main.obj as output)
> cobc -x -o prog main.obj subrs.obj
```

This creates a single executable file 'main.exe' in the ./object folder.

Building a shared library from COBOL and "C" routines

Example: Building a shared library combining COBOL and "C" routines

In Linux/Unix:

```
> cobc -c subr1.cob
> cobc -c subr2.cob
> cobc -c subr3.c
> cobc –b -o libsubrs.so subr1.o subr2.o subr3.o
```

In Windows:

```
> cobc -c subr1.cob
> cobc -c subr2.cob
> cobc -c subr3.c
> cobc –b -o subrs.dll subr1.obj subr2.obj subr3.obj
```

Linking a shared library with your main program

Example: Using a shared library by linking it with your main program
In Linux/Unix:

Before linking the library, install it in your system library directory:

```
> cp libsubrs.so /usr/lib
```

or install it somewhere else and set `LD_LIBRARY_PATH':

```
> cp libsubrs.so /your/COBOL/lib
> export LD_LIBRARY_PATH=/your/COBOL/lib
```

Then, compile the main program, linking the library as follows:

```
> cobc -x main.cob -L/your/COBOL/lib –lsubrs
```

In Windows:

In Windows, you need to place the shared library (.dll) in your PATH, and then use the link command to locate the .lib file that contains the stub for linking. Link the library, as follows:

```
> cobc -x main.cob –LC:\your\COBOL\lib –lsubrs.lib
```

Using the COBOL-IT Debugger

The COBOL-IT Debugger (cobcdb) has been designed to operate as an engine, working in the background, behind a user interface, such as Deet, or such as the interface that is provided by the COBOL-IT Debugging Perspective in the Developer Studio. The COBOL-IT Debugger (cobcdb) runs shared object files that have been created by the COBOL-IT Compiler (cobc) and that have been compiled with the –g compiler flag.

Conventions Used

The Debugger Prompt

When you start the COBOL-IT Debugger, the COBOL-IT Debugger Window presents a prompt, into which a Debugger Command can be entered. After entering a Debugger Command, the user will see the results of their command returned, with a subsequent debugger prompt. The default debugger prompt is (cobcdb).

To illustrate:

```
C:\COBOL\CobolIT\samples>cobcdb hello
CreateProcess "cobcrun -d hello ".
command:11516
(cobcdb)
event:11516
-event-end-stepping-range #0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!8
```

(The debugger prompt is here. As an example, enter the version command:)

```
(cobcdb)
version
~"COBOL-IT cobcdb 3.6.4\n"
^done
(cobcdb)
```

(Enter a subsequent command here.)

Source Location

Source Location is formatted as:

```
<Absolute source path name>!<line number>
```

Example:

```
C:/COBOL/CobolIT/samples/hello.cbl!21
```

Variables names

<variable-name> is formatted as:
`[@<module-name>.][<section>.][<upper-level-fields >.]<field-name>`

If no <module-name> is given, current module is searched
If no <section> is given, sections are searched in the following order: file section, working-storage section, linkage-section.
If no <upper-level-field> is given, the first matching field as presented in the original source is returned

Example:
`WORKING-STORAGE.WrkA.Wrk_G1.Wrk_G1_F1 or Wrk_G1.Wrk_G1_F1`

is equivalent to
`@PrgA.WORKING-STORAGE.WrkA.Wrk_G1.Wrk_G1_F1`

where declarations are:
working-storage section.
01 WrkA.
03 Wrk_F1 PIC 99.
03 Wrk_F2 PIC 99.
03 Wrk_G1.
05 Wrk_G1_F1 PIC 99.
05 Wrk_G1_F2 PIC 99.

Usage of the COBOL-IT Debugger:

```
>cobcdb [options] [program name]  [command-line parameters]
```

command-line parameters

are parameters which would be returned to the program through an `ACCEPT from COMMAND-LINE` statement.

program name

is the name of the shared object file created by the COBOL-IT Compiler (.dll, .so).

options

are parameters that are passed to the COBOL-IT Debugger. These options include:

-listdid

Causes the COBOL-IT Debugger to list all the running processes by PID, as well as debug-id. As an example:

```
C:\Cobol\CobolIT>cobcdb -listdid
did: --------   pid:    11412   module:
did: --------   pid:    11956   module:
did:    12345   pid:    11536   module: hello
did: --------   pid:     3296   module:
did: --------   pid:     3324   module:
```

-m

(Unix/Linux only). Disables ability to use up/down keys to return history of previous commands, and left/right arrows to edit the line. The –m functionality can be duplicated when running the Debugger, and using the command:

```
Set readline off
```

-n

(Windows only). Causes the COBOL-IT Debugger to start the execution of ***program name*** in a new cmd.exe window.

-p <did>

Causes the COBOL-IT Debugger to connect to the running process identified by *did*. *did* is the debug-id. Did may be a debug-id, set with the runtime environment variable COB_DEBUG_ID, or it may be the process id (pid) of the currently running process.
When using the **-p** *did* parameter, there is no need to specify ***program name***, as the program is Identified by *did*.

-r host:port

Connects two TCP sockets to *host:port*. Debugger commands, and the results returned are transmitted via these sockets. Used by the Remote System Explorer in the COBOL-IT Developer Studio.

Sockets are identified by the first line sent.

Socket1 is used to exchange Command/Result information. As an example, the COBOL-IT Debugger will READ Commands on Socket1, and WRITE the Results of the Command to that socket.

Socket1 is identified by "*command:pid*\n" where pid is the process-ID.
Socket2 is used to write Debugger Events. For more information about Debugger Events, see the Chapter below titled "Debugger Events".
Socket2 is identified by "event:pid\n" where pid is the process-ID.

-trace

Causes the COBOL-IT Debugger to write tracing information to cobcdb.out.

-w <did>

Causes the COBOL-IT Debugger to interrupt the process identified by *did* and set it into a "wait for connect" state. *did* is the debug-id. Did may be a debug-id, set with the runtime environment variable COB_DEBUG_ID, or it may be the process id (pid) of the currently running process. A program that has been set into this state can be debugged with the –p *did* command. When using the **-w** *did* parameter, there is no need to specify ***program name***, as the program is identified by *did*.

-y tty

(UNIX/Linux only). Causes the COBOL-IT Debugger to assign stdout/stdin/stderr to *tty*. When running the COBOL-IT Debugger with –y *tty*, ***program name*** is required.

Debugger Commands

Note- Abbreviations for the Debugger Commands are recognized by cobcdb. Where applicable, you will see the abbreviated version of the command listed in parentheses, after the debugger command.
Example:
break (br) This indicates that the abbreviation "br" is recognized as a synonym of break by cobcdb.

Debugger Commands can be repeated by using the [Enter] key. As an example, Single-stepping through a program can be done by entering the S command once, and then repeatedly hitting the [Enter] key.

Debugger Commands include:

break (br)

causes a breakpoint to be set in the location that is indicated. With the addition of the -t flag, breakpoints can be created as temporary breakpoints, which are erased after they have been reached the first time. The break command requires a location parameter. Location parameters for the break command are:

Module	Sets a breakpoint in a module, as identified by program-id.
label	Sets a breakpoint at a paragraph name.
line-nr	Sets a breakpoint at a line number.

module, label, and line-nr can be combined, with a ! notation.

break [-t] label

sets a breakpoint at a paragraph name .
Example:

```
(cobcdb)
break -t para-1
Breakpoint 1 in para-1 at C:/COBOL/CobolIT/samples/hello.cbl
(cobcdb)
```

break [-t] module!label

sets a breakpoint at a paragraph name (label) in a module. module is identified by source file name. If no module name is specified, then the current module is used. Since module may not be loaded yet, no validation of module!label is made.
Example:

```
(cobcdb)
break -t C:/COBOL/CobolIT/samples/hello.cbl!para-1
Breakpoint 2 in para-1 at C:/COBOL/CobolIT/samples/hello.cbl
(cobcdb)
```

Example: (sets a breakpoint at the entry to para-1 in the current source module)

```
(cobcdb)
break !para-1
Breakpoint 2 in para-1 at C:/COBOL/CobolIT/samples/hello.cbl
(cobcdb)
```

break [-t] module!line-nr

sets a breakpoint at a line number in a module. module is identified by source file name. if no module name is specified, then the current module is used. Since module may not be loaded yet, no validation of module!line-nr is made.

Example:

```
(cobcdb)
break -t C:/COBOL/CobolIT/samples/hello.cbl!22
Breakpoint 3 at C:/COBOL/CobolIT/samples/hello.cbl!22
(cobcdb)
```

Example: (Sets a breakpoint at line 11 of the current source module.)

```
(cobcdb)
break !11
Breakpoint 1 at /home/cobolit/hello.cbl ! 11
(cobcdb)
```

break [-t] module!0

sets a breakpoint at the entry-point to module. module is identified by source file name. if no module name is specified, then the current module is used.

Example:

```
break -t c:/cobol/cobolit/samples/subpgm.cbl!0
```

```
Breakpoint 1 at c:/cobol/cobolit/samples/subpgm.cbl ! 0
(cobcdb)
```

Or

```
break -t subpgm.cbl!0
Breakpoint 1 at subpgm.cbl ! 0
(cobcdb)
```

bt

causes a CALL/PERFORM stack trace to be generated. The format for the stack trace display is : #<frame-number><module>() at <source-location>

Example:

```
bt
#0 hello () at C:/COBOL/CobolIT/samples/hello.cbl!21
#1 hello () at C:/COBOL/CobolIT/samples/hello.cbl!16
(cobcdb)
```

frame-number 0 is the current program position

continue

causes execution of program to be continued until the next breakpoint is encountered, or until the end of the program . An event-continue command is issued. As seen in the example below, this is interrupted when an event-breakpoint-hit event takes place.

Example:

```
break -t para-1
Breakpoint 1 in para-1 at C:/COBOL/CobolIT/samples/hello.cbl
(cobcdb)
continue
-event-continue
-event-breakpoint-hit (cobcdb)#0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!22
(cobcdb)
```

Example :

```
break -t C:/COBOL/CobolIT/samples/hello.cbl!22
Breakpoint 1 at C:/COBOL/CobolIT/samples/hello.cbl ! 22
(cobcdb)
continue
-event-continue
-event-breakpoint-hit (cobcdb)#0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!22
```

contreturn

causes execution to continue to the next PERFORM return, or break on the first breakpoint reached, which ever comes first. An event-contreturn command is issued. This is interrupted when an –event-end-stepping-range event takes place.

Example :

```
contreturn
-event-contreturn
(cobcdb)-event-end-stepping-range #0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!17
```

delete (d) <x>

causes breakpoint number *x* to be deleted.

Example:

```
(cobcdb)
delete 3
^done
(cobcdb)
```

frame (f) <frame-number>

Prints the source location for the designated frame number. The frame numbers of an application run session are the points at which the application has branched either due to a PERFORM <paragraph> statement or a CALL <subprogram> statement.

Example:

```
(cobcdb)
frame 0
#0 hello () at C:/COBOL/CobolIT/samples/hello.cbl!25
(cobcdb)
frame 1
#1 hello () at C:/COBOL/CobolIT/samples/hello.cbl!17
(cobcdb)
```

info (i)

causes information to be displayed about the <info parameter> that is indicated.
The info command requires an <info parameter>.

Info parameters for the info command are:

break	Displays a list of breakpoints
locals	Displays a dump of the current variables in memory
sources	Displays a list of source files corresponding to loaded modules.

target Displays the Process ID of the runtime session.

info break

displays a list of breakpoints.

Example:

```
(cobcdb)
info break
Breakpoint 1 at /home/cobolit/hello.cbl ! 9
(cobcdb)
```

info locals

displays a dump of the values of the fields in the modules currently loaded in memory.

Example :

```
(cobcdb)
info locals
@hello.WORKING-STORAGE
                @hello.WORKING-STORAGE.RETURN-CODE =
[10]"+000000000"
                @hello.WORKING-STORAGE.TALLY = [10]"+000000000"
                @hello.WORKING-STORAGE.SORT-RETURN =
[10]"+000000000"
                @hello.WORKING-STORAGE.NUMBER-OF-CALL-PARAMETERS =
[10]"+000000000"
                @hello.WORKING-STORAGE.message-line = [11]"
"
                @hello.WORKING-STORAGE.ws-dummy = [1]" "
                @hello.WORKING-STORAGE.ctr = [6]"000000"
                @hello.WORKING-STORAGE.COB-CRT-STATUS = [4]"0000"
(cobcdb)
```

Info is returned in a structured tree using SECTION as a header in the form :
<variable name> = [<size>]”<string>”
<variable name> is the full qualified variable name
<size> is the number of characters in the string
<string> is the data in human readable form. Strings may contain null characters.

info sources

displays source files associated with objects loaded in memory

Example:

```
(cobcdb)
info sources
```

```
Source files
C:/COBOL/CobolIT/samples/hello.cbl
(cobcdb)
```

info target

displays the pid of the currently running process.

Example:

```
 (cobcdb)
info target
Child PID 19012
(cobcdb)
```

kill

kills the current process.

Example:

```
(cobcdb)
kill
-event-program-exited (cobcdb)#0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!
10
```

list (l)

displays source of the current module. Note- The source of the current module is also the source of the current stack frame, as stack frames are organized by module.
Source code displays use the following conventions:

Column 1	.	Followed by a Line Number
Columns 2-8		Line Number of the Source file
Column 9	[c]	Character. Possible values are:
	.	Regular line of source code
	>	Current line of execution
	*	Defined breakpoint

Example:

```
.0000007.          procedure division.
.0000008>          main.
.0000009*             display "hello world" line 10 col 10.
.0000010.              accept ws-dummy line 10 col 30.
.0000011.              stop run.
```

list [start-line [end-line]]

Acceptable values for [start-line] are:

[No value]	The current line of execution is the line of reference by default. 5 lines before, and 5 lines after the current line of execution are displayed.
XXXX	Changes the line of reference to XXXX. 5 lines before and 5 lines after line XXXX are displayed.
*	Causes all lines from the beginning of the source file to [end-line] to be displayed. If there is no [end-line], all lines of the source file from beginning to end are displayed.
-XXXX	Begin source display XXXX lines before the current line of execution.
+XXXX	Begin source display 5 lines before the current line of execution and end source display XXXX lines after the current line of execution.

Acceptable values for [end-line] are:

[No value]	Sets end-line to XXXX.
XXXX	Changes the line of reference to XXXX. 5 lines before and 5 lines after line XXXX are displayed.
*	Continue source display to the end of the source file.
+XXXX	End source display XXXX lines after the start line.

Examples:
list (no parameters)
In this case, the display will show 5 lines before and 5 lines after the current line of execution.

```
(cobcdb)
list
.0000003.        environment division.
.0000004.        data division.
.0000005.        working-storage section.
.0000006.        77 ws-dummy  pic x.
.0000007.        procedure division.
.0000008>        main.
.0000009.            display "hello world" line 10 col 10.
.0000010.            accept ws-dummy line 10 col 30.
.0000011.            stop run.
```

list (start-line, no end-line)

```
(cobcdb)
list 6
.0000001.        identification division.
.0000002.        program-id.  hello.
```

```
.0000003.         environment division.
.0000004.         data division.
.0000005.         working-storage section.
.0000006.         77 ws-dummy  pic x.
.0000007.         procedure division.
.0000008>         main.
.0000009.             display "hello world" line 10 col 10.
.0000010.             accept ws-dummy line 10 col 30.
.0000011.             stop run.
(cobcdb)
```

list (start-line, end-line)

```
(cobcdb)
list 9 10
.0000009.             display "hello world" line 10 col 10.
.0000010.             accept ws-dummy line 10 col 30.
(cobcdb)
```

list (start-line, *)

```
(cobcdb)
list 7 *
.0000007.         procedure division.
.0000008>         main.
.0000009.             display "hello world" line 10 col 10.
.0000010.             accept ws-dummy line 10 col 30.
.0000011.             stop run.
(cobcdb)
```

next (n)

causes execution to pass to the next statement- jumping over a CALL or PERFORM statement before breaking, unless the CALL'ed paragraph or PERFORM statement contains a breakpoint. An event-next command is issued. This is interrupted when an –event-end-stepping-range event takes place. The next command can be abbreviated as "n".

Example :

```
(cobcdb)
next
-event-next
-event-end-stepping-range (cobcdb)#0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!17
```

print <variable-name>

displays the value of the variable in human readable format.

Example:

```
print message-line
$1 = @hello.WORKING-STORAGE.message-line [11]"XXXXXXXXXXX"
(cobcdb)
```

The information returned is in the format:
$1=@module-name.section-name.variable-name[size]"[string]"
Where:

module-name is the program-id of the module being executed.
section-name is the section containing the variable being displayed.
size is the size, in bytes of the variable.
string is the contents of the variable in human-readable format.

printh <variable-name>

displays the value of the variable in hexadecimal format.

Example:

```
printh message-line
$1 = @hello.WORKING-STORAGE.message-line
[22]"58585858585858585858"
(cobcdb)
```

The information returned is in the format:
$1=@module-name.section-name.variable-name[size]"[string]"
Where:

module-name is the program-id of the module being executed.
section-name is the section containing the variable being displayed.
size is the size, in bytes of the variable.
string is the contents of the variable in hexadecimal format.

quit (q)

causes an exit from the debugger.

Example:

```
(cobcdb)
Quit
```

set

allows the user to set a <set parameter> to a different value.
The set command requires a <parameter>.
Parameters for the set command are:

prompt<prompt-string>	Sets the debugger prompt to <prompt-string>
readline [on | off]	Set to off to disable the use of the up/down arrow keys to return debugger command history, and

	to disable the use of the left/right arrow keys to navigate within the debugger line for editing purposes.
var <variable-name> <variable-value>	Sets the value of <variable-name>
varh <variable-name> <variable-value>	Sets the value of <variable-name> in hex notation

set prompt <prompt string>

sets the COBOL-IT Debugger prompt. The default setting for the COBOL-IT Debugger prompt is (cobcdb).

Example :

```
(cobcdb)
event:13556
-event-end-stepping-range #0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!9
set prompt >>>
>>>
```

set readline [on | off]

sets the ability to use the arrow keys on the debugger command-line. When readline is set on, up/down arrow keys can be used to retrieve debugger command history, and left/right arrow keys can be used to move the cursor for purposes of editing.

Example :

```
(cobcdb)
set readline off
(cobcdb)
set readline on
(cobcdb)
```

set var <variable-name> <variable-value>

sets variable content for variable-name to variable-value. Values are converted to the appropriate type. A number stored in a PIC 999 field will be converted before storing.

Example :

```
(cobcdb)
set var message-line "hello hello"
$1 = @hello.WORKING-STORAGE.message-line [11]"hello hello"
(cobcdb)
```

set varh <variable-name> <variable-value-hex>

sets variable content for variable-name to variable-value-hex. <variable-value-hex> must be a valid hexadecimal string. Note that in a valid hexadecimal string, a single character space is recorded with two characters, so the total string length of <variable-value-hex> must be exactly two times the length of <variable-name>.

Example:

```
(cobcdb)
set varh ws-dummy 41
$1 = @hello.WORKING-STORAGE.ws-dummy [1]"A"
(cobcdb)
```

step (s)

causes execution of the program to execute a single step, and then break. An event-step command is issued. This is interrupted when an –event-end-stepping-range event takes place. The step command can be abbreviated as “s”.

Example:

```
(cobcdb)
step
-event-step
(cobcdb)-event-end-stepping-range #0 hello () at
C:/COBOL/CobolIT/samples/hello.cbl!14
```

stop

causes execution to stop (break) at the next statement

up (u)

changes the current frame. When you have several levels of CALLs, the **info** functions relate to the current module.

up -[n]

In a CALL’ed subprogram, **up –[n]** can be used to change the frame back to a previous CALL’ing module. **Info locals** can then be viewed for that calling module.
In the example below, the **bt** command shows 3 frames, with frame 0 being the current frame in a called sub-program, and the **info locals** command showing the state of the variables in the subprogram. **up -1** sets the frame to the calling program, so that **info locals** can be viewed for the calling program.

```
bt
#0 subpgm () at C:/COBOL/CobolIT/samples/subpgm.cbl!7
```

```
#1 hello () at C:/COBOL/CobolIT/samples/hello.cbl!25
#2 hello () at C:/COBOL/CobolIT/samples/hello.cbl!17
(cobcdb)
info locals
@subpgm.WORKING-STORAGE
        @subpgm.WORKING-STORAGE.RETURN-CODE = [10]"+000000000"
        @subpgm.WORKING-STORAGE.TALLY = [10]"+000000000"
        @subpgm.WORKING-STORAGE.SORT-RETURN = [10]"+000000000"
        @subpgm.WORKING-STORAGE.NUMBER-OF-CALL-PARAMETERS =
[10]"+000000000"
        @subpgm.WORKING-STORAGE.COB-CRT-STATUS = [4]"    "
(cobcdb)
up -1
#1 hello () at C:/COBOL/CobolIT/samples/hello.cbl!25
(cobcdb)
info locals
@hello.WORKING-STORAGE
        @hello.WORKING-STORAGE.RETURN-CODE = [10]"+000000000"
        @hello.WORKING-STORAGE.TALLY = [10]"+000000000"
        @hello.WORKING-STORAGE.SORT-RETURN = [10]"+000000000"
        @hello.WORKING-STORAGE.NUMBER-OF-CALL-PARAMETERS =
[10]"+000000000"
        @hello.WORKING-STORAGE.message-line = [11]"XXXXXXXXXXX"
        @hello.WORKING-STORAGE.ws-dummy = [1]" "
        @hello.WORKING-STORAGE.ctr = [6]"000000"
        @hello.WORKING-STORAGE.COB-CRT-STATUS = [4]"0000"
(cobcdb)
```

version (v)

returns the version of the cobcdb/COBOL-IT runtime.

Example:

```
(cobcdb)
version
~"COBOL-IT cobcdb 3.6.4\n"
^done
(cobcdb)
```

Debugger Events

-event-breakpoint-hit

is returned when a breakpoint is hit.

-event-continue

is returned by the continue command. Terminated by –event-breakpoint-hit.

-event-contreturn

is returned by the contreturn command. Terminated by –event-end-stepping-range.

-event-end-stepping-range

is returned when one of the debugger step commands (step, next, contreturn) reaches the end of its stepping range.

-event-next

is returned by the next command. Terminated by –event-end-stepping-range.

-event-program-exited

is returned by the kill command.

-event-step

is returned by the step command. Terminated by –event-end-stepping-range.

Our Sample Programs

For the purposes of this documentation, we are using a very short hello.cbl program as a reference. (The program contains an ACCEPT FROM COMMAND-LINE statement, to illustrate this functionality in cobcdb.)

To compile: >cobc –g hello.cbl
>cobc –g subpgm.cbl
To run: >cobcdb hello (or)
To run with parameters: >cobcd hello hello-world

hello.cbl

```
000001 identification division.
000002 program-id.  hello.
000003 environment division.
000004 data division.
000005 working-storage section.
000006 77 message-line pic x(11) value spaces.
000007 77 ws-dummy   pic x value spaces.
000008 77 ctr        pic 9(6) value 0.
000009 procedure division.
000010 main.
000011    accept message-line from command-line.
```

```
000012     if message-line not = spaces
000013       display message-line line 10 col 10
000014     else
000015       display "hello world" line 10 col 10
000016     end-if.
000017     perform para-1.
000018     display "returned from para-1" line 14 col 10.
000019     display "next line"  line 16 col 10.
000020     accept ws-dummy line 16 col 30.
000021     stop run.
000022 para-1.
000023     move all "X" to message-line.
000024     display "in para-1" line 12 col 10.
000025     call "subpgm".
```

subpgm.cbl

```
000001 identification division.
000002 program-id. subpgm.
000003 environment division.
000004 data division.
000005 working-storage section.
000006 procedure division.
000007 main.
000008     display "In Subpgm" line 20 col 10.
000009     goback.
```

Interoperability Topics

COBOL/C Interoperability

Calling COBOL from C

Overview of key COBOL-IT API functions

`#include <libcob.h>`	libcob.h is located in $COBOLITDIR/include directory. Required in C main program.

`COB_RTD = cob_get_rtd();`	`COB_RTD is a macro that defines the runtime data rtd variable.`
`cob_init(rtd, 0, NULL);`	`Cob_init initializes the runtime. Pass 0, NULL if there are no parameters to pass to the runtime. Otherwise, argc, argv.`
`cob_stop_run (rtd, return_status);`	`Cleanup and terminate. This does not return.`

Calling COBOL programs

Function Prototypes

Case 1- The COBOL program "say.cbl" has two parameters described in the Linkage Section. In "C", this is equivalent to a function having the following prototype:

```
extern int say(char *hello, char *world);
```

Case 2- If you specified a PROGRAM-ID that is different from the source base name, two symbols will be generated. One of the symbols generated will use the PROGRAM-ID, and one will use the source base name. Expanding on the case above, if we change the PROGRAM-ID for say.cbl to MYSAY, as below,

say.cbl

IDENTIFICATION DIVISION.
PROGRAM-ID. MYSAY.
...

This would be the equivalent to a function having the following prototype:

```
extern int say(char *hello, char *world);
extern int MYSAY(char *hello, char *world);
```

Either of these functions can be called from the "C" program, as they both point to the same COBOL program.

Declaring a function prototype for a COBOL program with two parameters in the linkage section, and CALLing that COBOL program.

`extern int say(char *hello, char *world);` `ret = say(hello, world);`	Call a COBOL program "say", and pass two parameters into linkage data items.

Using cob_resolve to find a COBOL program

In the absence of a function prototype, you can find a COBOL module having a specific PROGRAM-ID by CALLing the function cob_resolve. There is an example of this usage in the sample hello-dynamic.c below.

`say = cob_resolve(rtd, "say")`
`if (say == NULL) {`
`fprintf(stderr, "%s\n",cob_resolve_error (rtd));`
`exit(1);`
`}`
`ret = say(hello, world);`
	cob_resolve takes the module name as a string and returns a pointer to the module function.
	cob_resolve returns NULL if there is no module.
	cob_resolve_error can be called to return the error message.
	If the module say is found, it is called and passed two parameters.

This chapter describes how to interface C programs and routine with COBOL-IT programs, statically or dynamically.

Writing the Main Program in C

Examples follow, with cases where “C” programs are statically linked with COBOL programs, and cases where “C” programs are dynamically linked with COBOL programs.

Static linking of “C” programs with COBOL programs

The “C” program

```
/* hello.c */
#include <libcob.h>
extern int say(char *hello, char *world);
int main()
{
COB_RTD = cob_get_rtd();
int ret;
int return_status;
char hello[7] = "Hello ";
char world[7] = "World!";
cob_init(rtd, 0, NULL);
ret = say(hello, world);
cob_stop_run (rtd, return_status);
return ret;
}
```

Compile the "C" program

In Linux/Unix
```
>cc -c `cob-config --cflags` hello.c
```

In Windows
```
>cobc –c hello.c
```

The COBOL program

```
Say.cbl is passed two fields, which are described in the Linkage
Section.  Say.cbl DISPLAYs the two fields, and then exits.

say.cbl

IDENTIFICATION DIVISION.
PROGRAM-ID. say.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 HELLO PIC X(6).
01 WORLD PIC X(6).
PROCEDURE DIVISION USING HELLO WORLD.
DISPLAY HELLO WORLD.
EXIT PROGRAM.
```

Compile the COBOL program
In Linux/Unix and Windows:
```
>cobc -c -static say.cbl
```

Statically link the "C" and COBOL programs
In Linux/Unix:
```
>cobc -x -fno-main -o hello hello.o say.o
```

In Windows:
```
>cobc -x –flink-only -o hello hello.obj say.obj
```

Run the linked executable
In Linux/Unix:
```
>./hello
```

In Windows:
```
>hello
```

In summary

You can combine the compile and run commands above into scripts (Linux/Unix) or batch

files (Windows) as follows:

Linux/Unix

```
>cc -c `cob-config --cflags` hello.c
>cobc -c -static say.cbl
>cobc -x -fno-main -o hello hello.o say.o
>./hello
```

Windows 32, Windows 64

```
>cobc -c hello.c
>cobc -c -static say.cbl
>cobc -x -flink-only -o hello hello.obj say.obj
>hello
```

Running hello returns the following output:

```
Hello World!
```

Dynamic linking of "C" programs with COBOL programs

Note- This sample contains usage of the functions cob_resolve( ) and cob_resolve_error, which can be used to locate a COBOL module/report errors. The "C" program is compiled to an executable, and COBOL program is compiled to a separate shared object (DLL in Windows). COB_LIBRARY_PATH is set, and the CALL of the COBOL program is resolved dynamically.

The "C" program

```
/* hello-dynamic.c */
#include <libcob.h>
static int (*say)(char *hello, char *world);
int main()
{
/* COBOL-Runtime data */
/* COB_RTD is a macro that define rtd variable*/
COB_RTD = cob_get_rtd();
int ret;
char hello[7] = "Hello ";
char world[7] = "World!";
cob_init(rtd, 0, NULL);
/* find the module with PROGRAM-ID "say". */
say = cob_resolve(rtd, "say");
/* if there is no such module, show error and exit */
if (say == NULL) {
fprintf(stderr, "%s\n", cob_resolve_error (rtd));
exit(1);
}
/* call the module found and exit with the return code */
ret = say(hello, world);
```

```
return ret;
}
```

Compile the "C" program

In Linux/Unix

```
>cc -c `cob-config --cflags` hello-dynamic.c
>cobc -x –o hello hello-dynamic.o
```

In Windows

```
>cobc –x –flink-only –o hello hello-dynamic.c
```

The COBOL program

Say.cbl is passed two fields, which are described in the Linkage Section. Say.cbl DISPLAYs the two fields, and then exits.

say.cbl

```
IDENTIFICATION DIVISION.
PROGRAM-ID. say.
ENVIRONMENT DIVISION.
DATA DIVISION.
LINKAGE SECTION.
01 HELLO PIC X(6).
01 WORLD PIC X(6).
PROCEDURE DIVISION USING HELLO WORLD.
DISPLAY HELLO WORLD.
EXIT PROGRAM.
```

Compile the COBOL program

In Linux/Unix and Windows:

```
>cobc -m say.cbl
```

Dynamically link the "C" and COBOL programs
In Linux/Unix:

```
>export COB_LIBRARY_PATH=.
>./hello
```

In Windows:

```
>set COB_LIBRARY_PATH=.
>hello
```

Exiting COBOL, Returning to "C"

A COBOL main program can be written with an "Exit Program" statement, causing the program to return to the calling "C" program.

This is done by setting the exit-program-forced Compiler Configuration flag.

To cause the "EXIT PROGRAM" statement to return control to a calling "C" program, add the compiler figuration flag:

Exit-program-forced:yes

For more detail, see the explanation below:

exit-program-forced

- The exit-program-forced configuration file flag changes the way that the EXIT PROGRAM statement is handled.

If set to no (default) the program is exited only if it is not the main program.
If set to yes the EXIT PROGRAM verb always exits the current program.

In summary

You can combine the compile and run commands above into scripts (Linux/Unix) or batch files (Windows) as follows:

Linux/Unix

```
>cc -c `cob-config --cflags` hello-dynamic.c
>cobc -x -o hello hello-dynamic.o
>cobc -m say.cbl
>export COB_LIBRARY_PATH=.
>./hello
```

Windows 32, Windows 64

```
>cobc -x -flink-only -o hello hello-dynamic.c
>cobc -m say.cbl
>set COB_LIBRARY_PATH=.
>hello
```

Running hello returns the following output:

```
Hello World!
```

Note- if the COBOL program say.cbl has not been compiled or if COB_LIBRARY_PATH is not set correctly, then running hello will produce the output:

```
Cannot find module 'say'
```

Calling C from COBOL

CALL'ing a "C" program from a COBOL program does not require any special coding convention.

Please note, however, that unlike C, text arguments passed as parameters from COBOL are not terminated by the null character (i.e., `\0'). Hence, the CALL'ed "C" function cannot rely on the existence of this termination character– unless the COBOL code has been specifically written with interfacing with C in mind, and the termination characters have been set explicitly. Note that the example below uses hard-coded text values, 6 characters in length. When passing data items in which text length can be variable, it can be helpful to pass length information in a separate parameters.

Note- Some of the sample "C" programs in this section are compiled with cobc.

Remember that `cobc` translates COBOL into "C", and then invokes the local "C" compiler. When cobc detects that the target file is written in "C", it skips the preprocessing and translation steps, and proceeds directly to invoking the host "C" compiler with certain default settings.

Thus, the command "cobc –c say.c" differs from the command "cl –c say.c" int that cobc also applies certain default settings to the "C" compiler. There are times when this will be convenient, and times when it won't. The user always has the option of using their "C" compiler to compile the "C" programs in these samples. But, it is important to understand that the commands "cobc –c" and "cl –c" (Windows) are not equivalent, as the use of cobc does apply additional compiler flags.

For a full explanation of what takes place when you use the command "cobc –c say.c", see the Appendix topic **Compiling a "C" program with cobc** .

Static linking COBOL programs with C programs

The COBOL program

```
IDENTIFICATION DIVISION.
PROGRAM-ID. Hello.
ENVIRONMENT DIVISION.
DATA DIVISION.
WORKING-STORAGE SECTION.
01 HELLO PIC X(6) VALUE "Hello ".
01 WORLD PIC X(6) VALUE "World!".
PROCEDURE DIVISION.
CALL "say" USING HELLO WORLD.
STOP RUN.
```

Compile the COBOL program
In Linux/Unix, and Windows

```
>cobc -c -static hello.cbl
```

The -static command-line option ensures that calls will be translated to plain static C function calls.

Note that when CALL'ing a "C" routine from a COBOL program, case-sensitivity must be respected. That is, if you were to try to reproduce this sample, while not respecting case-sensitivity, with a statement CALL "SAY" using HELLO, WORLD., then when you linked the two object files together into the executable "hello.exe", you would receive an "Unresolved external symbol" error at link time:

hello.obj : error LNK2019: unresolved external symbol SAY referenced in function
Hello_
hello.exe : fatal error LNK1120: 1 unresolved externals

To resolve this, you would have to change the CALL statement, to CALL "say"….

The "C" program

Note- For this sample the Unix versions and Windows versions of say.c differ in the function prototype declaration.

In Unix:

```
int say(char *hello, char *world)
```

In Windows:

```
__declspec(dllexport)  int say(char *hello, char *world)
```

```
/* say.c Unix Version */
int say(char *hello, char *world)
{
int i;
for (i = 0; i < 6; i++)
putchar(hello[i]);
for (i = 0; i < 6; i++)
putchar(world[i]);
putchar('\n');
return 0;
}

/* say.c Windows Version */
__declspec(dllexport) int say(char *hello, char *world)
{
int i;
for (i = 0; i < 6; i++)
putchar(hello[i]);
for (i = 0; i < 6; i++)
putchar(world[i]);
putchar('\n');
return 0;
}
```

Compile the "C" program
In Linux/Unix
```
>cc -c say.c
```

In Windows
```
>cobc –c say.c
```

Statically link the COBOL and "C" programs
In Linux/Unix:
```
>cobc -x -o hello hello.o say.o
```

In Windows:
```
>cobc -x -o hello hello.obj say.obj
```

Run the linked executable
In Linux/Unix:
```
>./hello
```

In Windows:
```
>hello
```

In summary

You can combine the compile and run commands above into scripts (Linux/Unix) or batch files (Windows) as follows:

Linux/Unix
```
>cobc -c -static hello.cbl
>cc -c say.c
>cobc -x -o hello hello.o say.o
>./hello
```

Windows 32, Windows 64
```
>cobc -c -static hello.cbl
>cobc –c say.c
>cobc -x -o hello hello.obj say.obj
>hello
```

```
Note- For more information on using cobc to compile "C" programs,
see the Appendix
```

Running hello returns the following output:

```
Hello World!
```

Dynamic linking COBOL programs with C programs

You can call a C shared library from a COBOL-IT program. Begin by compiling your C module(s) into a shared library rather than a static object file, do not use the -static command line option. This

will ensure that the COBOL-IT runtime will find the shared library for you.

This sample is also designed to demonstrate a case where "C" functions are CALL'ed with different parameter types. Note in the example below how "say" is called using the character strings "Hello" and "World!", and then called again using the integers Val1 and Val2. See "say.c" for details on how this sort of case is handled within the "C" program.

The COBOL program

```
       IDENTIFICATION DIVISION.
       PROGRAM-ID. Hello.
       ENVIRONMENT DIVISION.
       DATA DIVISION.
       WORKING-STORAGE SECTION.
       01 HELLO PIC X(6)  VALUE "Hello ".
       01 WORLD PIC X(6)  VALUE "World!".
       01 VALARE PIC X(9) VALUE "Value is ".
       01 VAL1 PIC 9(3)   VALUE 10.
       PROCEDURE DIVISION.
       MAIN.
       CALL "say" USING HELLO WORLD.
       CALL "say" USING VALARE VAL1.
       STOP RUN.
```

Compile the COBOL program
In Linux/Unix, and Windows

```
>cobc -x hello.cbl
```

The "C" program

```
/* say.c */
#include "libcob.h"
/* Use function type with COB_CALL_TARGET for been usable from COBOL */
/* On Windows COB_CALL_TARGET is defined as __declspec(dllexport)*/
COB_CALL_TARGET int say (char * data1 , char * data2)
{
    COB_RTD = cob_get_rtd();
    cob_field * f1;
    cob_field * f2;
    int i;
    if (rtd->cob_call_params != 2) {
        printf("Invalid parameter count %d ", rtd->cob_call_params);
        return 0;
    }
    f1 = rtd->current_module->cob_procedure_parameters[0];
    f2 = rtd->current_module->cob_procedure_parameters[1];
    if (COB_FIELD_TYPE(f1) != COB_TYPE_ALPHANUMERIC) {
        printf("Fisrt parameters must be Alphanumeric ");
        return 0;
    }
```

```
    for (i = 0; i < f1->size; i++)
        putchar(f1->data[i]);

    switch (COB_FIELD_TYPE(f2)) {
        case COB_TYPE_UNKNOWN:
        case COB_TYPE_GROUP :
/* COB_FIELD_DATA(f2) point to structure */
            break;
        case COB_TYPE_ALPHANUMERIC :
        case COB_TYPE_ALPHANUMERIC_ALL:
        case COB_TYPE_ALPHANUMERIC_EDITED :
        case COB_TYPE_NUMERIC_DISPLAY:
        case COB_TYPE_NUMERIC_EDITED :
/* COB_FIELD_DATA(f2) point to array of char */
            for (i = 0; i < f2->size; i++)
                putchar(f2->data[i]);
            break;
        case COB_TYPE_NUMERIC_BINARY:
/* COB_FIELD_DATA(f2) point to binary data size of */
/* f2->size */
/* if COB_FIELD_BINARY_SWAP(f) is true then bytes */
/* order are swapped regarding native platform byte order */
            break;
        case COB_TYPE_NUMERIC_PACKED:
/* COB_FIELD_DATA(f2) point to COMP-3 data of */
/* COB_FIELD_DIGITS(f2) digits and COB_FIELD_SCALE(f)
scale*/
/* if COB_FIELD_PACKED_SIGN_MISSING(f) is true then field */
/* is COMP-6*/
            break;
        case COB_TYPE_NUMERIC_FLOAT :
/* COB_FIELD_DATA(f2) point to a C float (may be not
aligned */
            break;
        case COB_TYPE_NUMERIC_DOUBLE:
/* COB_FIELD_DATA(f2) point to a C double */
/* (may be not aligned */
            break;
        case COB_TYPE_NATIONAL:
        case COB_TYPE_NATIONAL_EDITED:
/* COB_FIELD_DATA(f2) point to array of UTF16 16bits chars */
            break;
    }
    putchar('\n');
}
```

Note- For more detailed information on cob_field_data, reference libcob/common.h.

.

Compile the "C" program

In Linux/Unix

```
>cc -shared -o say.so say.c
```

In Windows

```
>cobc –m say.c
```

Dynamically link the COBOL and "C" programs
In Linux/Unix:

```
>export COB_LIBRARY_PATH=.
>./hello
```

In Windows:

```
>set COB_LIBRARY_PATH=.
>hello
```

In summary

You can combine the compile and run commands above into scripts (Linux/Unix) or batch files (Windows) as follows:

Linux/Unix

```
>cobc -x hello.cbl
>cc -shared -o say.so say.c
>export COB_LIBRARY_PATH=.
>./hello
```

Windows 32, Windows 64

```
>cobc -x hello.cbl
>cobc –m say.c
>set COB_LIBRARY_PATH=.
>hello
```

Running hello returns the following output:

```
Hello World!
```

COBOL/Java Interoperability

With COBOL-IT, calling COBOL from Java requires an intermediate "C" program, which in the sample below, we refer to as a "JNI glue" program. You can use JNI to call COBOL-IT DLLs in Windows or shared libraries that contain COBOL code routines in UNIX.

In effect, the way from Java to COBOL-IT is through "C", as suggested in this diagram:

Java < > "C" < > COBOL

The following example includes a Java program, a "JNI glue" program (written in "C") and a COBOL program, which receives the data passed from the original Java program, and processes it. This is only a working sample. Please refer to the Java JNI documentation for full detail about calling C code from Ja va.

Take care to match memory models (32-bit or 64-bit) between your Java and COBOL installations. Mixing 32-bit (Java or Cobol) with 64-bit (Java or Cobol) is not allowed.

Prerequisites:

The COBOL/Java interoperability samples require that the Java Development Kit (JDK) be installed on the host system. Visit the Oracle website for information on how to download and install the Java Development Kit.

This sample also requires that a "C" compiler, and COBOL-IT COBOL compiler be installed on the host system. Linux/Unix systems will typically have a "C" compiler installed. For Windows systems that do not have a "C" compiler installed, visit the Microsoft website for information on how to download and install a Visual Studio C++ compiler.

Calling COBOL from Java

Many businesses looking at Application Modernization will naturally look to Java for building graphical front ends, and enabling them to connect internet portals, mobile devices and application servers to their core application technology. With COBOL/Java Interoperability, solution engineers are able to propose solutions that include websites, connecting to application servers such as WebLogic, WebSphere, or JBoss, connecting in turn to legacy COBOL applications, which connect in turn to industry-leading database engines such as Oracle, DB2, Microsoft SQL Server, MySQL, and PostgreSQL.

While it is clear that Java is a powerful enabler in these cases, it is equally clear that it is not designed to serve as a replacement for the legacy COBOL applications. As a result, we see COBOL/Java Interoperability developing as an important topic in the area of Application Modernization.

In a Java/COBOL solution, the COBOL legacy applications continue to run the business- and are deployed as programs that are CALL'ed from Java through a "C" calling interface known as the JNI.

The Java Program

This is a Java program that will take an argument as input, and pass that information to the JNI glue program, written in "C". In our example, the JNI glue program is progjavainterface.so (Linux/Unix), or progjavainterface.dll (Windows).

JavaProg.java

```
public class JavaProg
{
  public native String prog(int i, String s);
  static {
      System.loadLibrary("progjavainterface");
  }

  public static void main(String[] args) {

      String s = "From Java";
      JavaProg cobol= new JavaProg();
```

```
        for (int i=0; i<args.length; i++) {
           s = cobol.prog(i, args[i]);
           System.out.println("JAVA: Returned : " + s);
        }
     }
  }
```

Verify that Java Development Kit (JDK) is installed on your machine, and that the bin directory of the JDK installation is in your PATH. Then, you can compile the java program as follows:

```
>javac JavaProg.java
```

The JNI Glue Program

To be able to call this COBOL program from our Java code we need a “JNI glue” program written in “C”.

progjavainterface.c

```
#include "jni.h"
#include "libcob.h"

 JNIEXPORT void JNICALL Java_JavaProg_prog
   (JNIEnv *env, jobject obj, jint javaint, jstring javaString)
 {
     /*Get the integer from javaint*/
     const int nativeint =  javaint;

     /*Get the native string from javaString*/
     const char *nativeString = (*env)->GetStringUTFChars(env,
javaString, 0);

     /*Get COBOL-IT runtime data*/
     cit_runtime_t * rtd;

     /*The COBOL program interface */
     int (*cobolprog)(int *, char*);

     /* Buffer to avoid memory protection in cobol this must be
the same size +1 as string declared in LINKAGE*/
     char cobolstring[31];
     strncpy (cobolstring, nativeString, 31);

     /*call the cobol program*/
     rtd = cob_get_rtd();
     cob_init(rtd, 0, NULL);
```

```
cobolprog = cob_resolve(rtd, "prog");
if(cobolprog) {
    cobolprog(&nativeint, cobolstring);
}

/*DON'T FORGET THIS LINE!!!*/
(*env)->ReleaseStringUTFChars(env, javaString, nativeString);
```

Compile the “JNI glue” program written in “C” and create a shared object:

In Linux/Unix:

```
cobc -b progjavainterface.c -o libprogjavainterface.so
```

Be careful to name the output lib<yourname> ; this is required by the Unix loader.

In Windows:

```
cobc -I "C:\Program Files\Java\jdk1.6.0_22\include" -I "C:\Program
Files\Java\jdk1.6.0_22\include\win32" -b progjavainterface.c
```

The COBOL-IT program

In our sample, the JNI glue program, written in “C”, has been hard-coded to call the COBOL program “prog”, which will load the shared object prog.so (Linux/Unix) or prog.dll (Windows).

prog.cbl

```
       IDENTIFICATION   DIVISION.
       PROGRAM-ID.      prog.
       ENVIRONMENT      DIVISION.
       DATA             DIVISION.
       LINKAGE SECTION.
       01 VALINT   USAGE UNSIGNED-INT.
       01 VALSTR PIC X(30) USAGE DISPLAY.
       01 RETS.
           03 RETSTR1 PIC 9(9)  USAGE DISPLAY.
           03 RETSTR2 PIC X(30) USAGE DISPLAY.
           03 FILLER PIC X(1) VALUE ZERO.
      *
       PROCEDURE DIVISION USING VALINT VALSTR RETS.
           DISPLAY "COBOL: " VALINT "  IS '" VALSTR "'".
           MOVE VALINT to RETSTR1.
           MOVE VALSTR to RETSTR2.
           GOBACK.
```

Compile prog.cbl

```
cobc -m -fthread-safe prog.cbl
```

Take care to use the -fthread-safe compiler flag with all of your COBOL programs, even those not directly called by Java.

Note that we use USAGE UNSIGNED-INT for the numeric input value in the LINKAGE section. This type matches with the native “C” int type.

Please refer to the COBOL-IT Reference manual for more detail about USAGE memory mapping.

Run the java program

In Linux/Unix:

```
run.sh
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
java JavaProg "12345" "Calling COBOL from Java" "c"
```

In Windows:

```
java -classpath . JavaProg "12345" "Calling COBOL from Java" "c"
```

In summary

You can combine the compile and run commands above into scripts (Linux/Unix) or batch files (Windows) as follows:

Linux/Unix
buildnrun.sh

```
javac JavaProg.java
cobc -b progjavainterface.c -o libprogjavainterface.so
cobc -m -fthread-safe prog.cbl
export LD_LIBRARY_PATH=.:$LD_LIBRARY_PATH
java JavaProg "12345" "Calling COBOL from Java" "c"
```

Windows 32
build.bat

```
set JAVA_HOME=c:\program files\java\jdk1.6.0_22\bin
set PATH=%JAVA_HOME%;%PATH%
javac JavaProg.java
cobc -I "C:\Program Files (x86)\Java\jdk1.6.0_20\include" -I
"C:\Program Files (x86)\Java\jdk1.6.0_20\include\win32" -b
progjavainterface.c
cobc -m -fthread-safe prog.cob
java -classpath . JavaProg "12345" "Calling COBOL from Java" "c"
```

Windows 64

buildx64.bat

```
set JAVA_HOME=c:\program files\java\jdk1.6.0_22\bin
set PATH=%JAVA_HOME%;%PATH%
javac JavaProg.java
cobc -I "C:\Program Files\Java\jdk1.6.0_22\include" -I "C:\Program
Files\Java\jdk1.6.0_22\include\win32" -b progjavainterface.c
cobc -m -fthread-safe prog.cob
java -classpath . JavaProg "12345" "Calling COBOL from Java" "c"
```

Running JavaProg returns the following output:

```
COBOL: 0000000000  IS '12345                          '
JAVA: Returned : 00000000012345
COBOL: 0000000001  IS 'Calling COBOL from Java        '
JAVA: Returned : 000000001Calling COBOL from Java
COBOL: 0000000002  IS 'c                              '
JAVA: Returned : 000000002c
```

Calling Java from COBOL

With COBOL-IT, calling Java from COBOL requires an intermediate “C” program, which in the sample below, we refer to as a “JNI glue” program. In effect, the way from COBOL-IT to Java is through “C”, as suggested in this diagram:

COBOL-IT < > “C” < > Java

The following example includes a COBOL-IT program, a “JNI glue” program (written in “C”) and a Java program, which receives the data passed from the original Java program, and processes it. This is only a working sample. Please refer to the COBOL-IT documentation for full detail about calling C code from COBOL-IT.

Take care to match memory models (32-bit or 64-bit) between your Java and COBOL installations. Mixing 32-bit (Java or Cobol) with 64-bit (Java or Cobol) is not allowed

The COBOL-IT Program

prog.cbl

```
       IDENTIFICATION   DIVISION.
       PROGRAM-ID.      prog.
       ENVIRONMENT      DIVISION.
       INPUT-OUTPUT     SECTION.
       FILE-CONTROL.
       DATA             DIVISION.
       WORKING-STORAGE SECTION.
       01 STRDATA.
         02 STR PIC X(20) VALUE "COBOL STRING".
         02 FILLER PIC X  VALUE LOW-VALUE.
```

```
       PROCEDURE        DIVISION.
           CALL "CALLJAVA" using STRDATA.
           STOP RUN.
```

Compile prog.cbl

```
cobc -x prog.cbl
```

The JNI Glue Program

CALLJAVA.c

```
#include <stdio.h>
#include <jni.h>
#include <string.h>
#include "libcob.h"
#define PATH_SEPARATOR ':' /* define it to be ';' on windows */
#define USER_CLASSPATH "." /* where Prog.class is */

JNIEnv* create_vm(JavaVM ** jvm) {

    JNIEnv *env;
    JavaVMInitArgs vm_args;
    JavaVMOption options;
    int ret;

    options.optionString = "-Djava.class.path=."; //Path to the
java source code
    vm_args.version = JNI_VERSION_1_6; //JDK version. This
indicates version 1.6
    vm_args.nOptions = 1;
    vm_args.options = &options;
    vm_args.ignoreUnrecognized = 0;

    ret = JNI_CreateJavaVM(jvm, (void**)&env, &vm_args);
    if (ret < 0)
        printf("\nUnable to Launch JVM\n");
    return env;
}


COB_CALL_TARGET int CALLJAVA(char* str)
{
    JNIEnv *env;
    JavaVM * jvm;
    jclass clsH=NULL;
    jmethodID midCalling = NULL;
    jstring StringArg;
```

```
    /* create Jave VM ... This should be done only once */
    env = create_vm(&jvm);
    if (env == NULL)
        return 1;

    // Find the Java Class
    clsH = (*env)->FindClass(env, "jmodule");

    //Obtaining Method IDs
    if (clsH != NULL) {
        midCalling    = (*env)->GetStaticMethodID(env,
clsH,"TestCall","(Ljava/lang/String;)V");
    } else {
        printf("\nUnable to find the requested class\n");
    }

    if (midCalling!=NULL) {
        StringArg = (*env)->NewStringUTF(env, str);

        //Calling another static method and passing string type
parameter
        (*env)->CallStaticVoidMethod(env,
clsH,midCalling,StringArg);
    }

    /* close Java VM*/
    (*jvm)->DestroyJavaVM(jvm);
}
```

Compile the “JNI glue” program written in “C” and create a shared object:

In Linux/Unix:

```
cobc -m -I $JAVA_HOME/include -I $JAVA_HOME/include/linux
CALLJAVA.c -R $JAVA_HOME/jre/lib/amd64/server -L
$JAVA_HOME/jre/lib/amd64/server -ljvm
```

In Windows:

```
SET JAVA_HOME=C:\Program Files (x86)\Java\jdk1.6.0_20
PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin\server;%PATH%
SET LIB=%LIB%;%JAVA_HOME%\lib
SET
INCLUDE=%INCLUDE%;%JAVA_HOME%\include;%JAVA_HOME%\include\win32
cobc -m CALLJAVA.c  -l jvm.lib
```

The Java Program

```
jmodule.java

public class jmodule
{
      public static void main(String args[])
      {
         System.out.println("Hello World!");
         System.out.println("This is the main function in jmodule
class");
      }
      public static void TestCall(String szArg)
      {
        System.out.println("Print from Java");
          System.out.println(szArg);
      }
}
```

Verify that Java Development Kit (JDK) is installed on your machine, and that the bin directory of the JDK installation is in your PATH.

Compile the java program:

```
>javac jmodule.java
```

Run the COBOL Program

In Linux/Unix

```
./prog
```

In Windows

```
SET JAVA_HOME=C:\Program Files\Java\jdk1.6.0_22
SET LIB=%LIB%;%JAVA_HOME%\lib
./prog
```

In summary

You can combine the compile and run commands above into scripts (Linux/Unix) or batch files (Windows) as follows:

Linux/Unix

```
cobc -x prog.cob
cobc -m -I $JAVA_HOME/include -I $JAVA_HOME/include/linux
CALLJAVA.c -R $JAVA_HOME/jre/lib/amd64/server -L
$JAVA_HOME/jre/lib/amd64/server -ljvm
```

```
javac jmodule.java
./prog
```

Windows 32

```
SET JAVA_HOME=C:\Program Files (x86)\Java\jdk1.6.0_22
SET PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin\server;%PATH%
SET LIB=%LIB%;%JAVA_HOME%\lib
SET
INCLUDE=%INCLUDE%;%JAVA_HOME%\include;%JAVA_HOME%\include\win32
cobc -x prog.cob
cobc -m CALLJAVA.c -l jvm.lib
javac jmodule.java
prog
```

Windows 64

```
SET JAVA_HOME=C:\Program Files\Java\jdk1.6.0_22
SET PATH=%JAVA_HOME%\bin;%JAVA_HOME%\jre\bin\server;%PATH%
SET LIB=%LIB%;%JAVA_HOME%\lib
SET
INCLUDE=%INCLUDE%;%JAVA_HOME%\include;%JAVA_HOME%\include\win32
cobc -x prog.cob
cobc -m CALLJAVA.c -l jvm.lib
javac jmodule.java
prog
```

Running Prog returns the following output:

```
Print from Java
COBOL STRING
```

IBM(R) DB2(R)

Due to the high level of compatibility with Micro Focus that COBOL-IT provides with its MF command line emulator "`cobmf`", current documentation provided by DB2 for use with Micro Focus COBOL can be used by COBOL-IT users, with just a few adjustments, documented below.

"cobmf"- the COBOL-IT MF Command-line Emulator

`cobmf`, or `cobmf.exe` in Windows environments, is located in the $COBOLITDIR\bin directory. For a full list of compiler flags supported by cobmf, just type cobmf [return] at the command line.

`cobmf` facilitates the transition from Micro Focus COBOL to COBOL-IT by providing a Micro Focus command-line emulator. The user can rename `cobmf` to `cob` (or rename `cobmf.exe` to `cob.exe` in Windows environments) , and continue to use the same compiler flags and

environment variables that they have developed over time. Establishing a link between cobmf and cob can also be useful:

```
>ln -s $COBOLITDIR/bin/cobmf $COBOLITDIR/bin/cob
```

Using citdb2.c

On some platforms, you may encounter a runtime error stating :

```
yourmodule.cbl:0: libcob: Cannot find module 'XXXXX'
```

Where XXXXX is a DB2 function called by your source.

To solve this you will need to add a 'fake' module to force the link of the function into your program. The COBOL-IT distribution includes the file $COBOLITDIR/lib/cobol-it/citdb2.c, which is intended to be used for this purpose. Citcb2.c includes a stub for the sqlgmf( ) function to serve as an example. You may expand this file to include stubs for all of the DB2 functions used in your COBOL program.

Including citdb2.c in a compile command is done as follows:

```
>cobc -x yourmod.cbl $COBOLITDIR/lib/cobol-it/citdb2.c
```

Source for citdb2.c

```
/*
 * Copyright (C) 2008 Cobol-IT
 *
 * This program is free software; you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation; either version 2, or (at your option)
 * any later version.
 *
 * This program is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with this software; see the file COPYING.  If not, write to
 * the Free Software Foundation, 51 Franklin Street, Fifth Floor
 * Boston, MA 02110-1301 USA
 */

/* This module is only a fake to use with DB2 Gmf library
 * to force a reference to the library.
 */
extern void sqlgmf(void);

/*
 * DO NEVER CALL this function... just link the module with your program using
 * DB2gmf  library
 */
void CIT_db2_stub(void)
{
        sqlgmf();
}
```

EXTFH

Overview

The *External File Handler* (EXTFH) is a publicly documented interface that allows COBOL applications to use indexed and sequential files that are EXTFH-compliant for record storage.

Either at compile-time, or at run-time, the COBOL-IT object can be informed that FILE I-O will be done using CALLs to EXTFH, and directed through an EXTFH library to an EXTFH-compliant data source.

The EXTFH library is passed a File Control Description (FCD) structure, and uses this information to interact with the data source. Data, and file status codes are returned to the COBOL program through the FCD. Updating of FD structures, and file status codes in the COBOL program is automatic.

In summary, the key elements in using EXTFH are :

- The EXTFH file handler. This is optional. The default EXTFH file handler is EXTFH. That is, when you elect to use EXTFH, your IO statements will be translated by default into CALLs to EXTFH. There exist a number of ways to substitute a file handler for EXTFH, as documented below.
- The EXTFH library. The EXTFH library is the library that is used by your file handler. The EXTFH library can be linked at compile-time. If is provided as a shared library, it must be in the COB_LIBRARY_PATH, or PATH in your runtime environment.
- The COBOL-IT user can set up their EXTFH interface in any of the following ways :
    - Configure EXTFH at compile time, using compiler flags
    - Configure EXTFH at compile time, using settings in the compiler configuration file
    - Configure EXTFH at run time, using runtime environment variables

Using the COBOL-IT EXTFH interface

Enable EXTFH using compiler flags

To enable the use of an EXTFH-compliant data source at compile time, add the compiler flag

-use-extfh= <file handler> to your command-line. :

Note: < file handler> is optional. The default value is EXTFH.

Compililing with –use-extfh causes the different file I-O statements to be translated at compile time, such that the fcd

When the -use-extfh compiler flag is used, all file i-o performed using standard COBOL verbs is redirected to a call of the external symbol <file handler>.

Enable EXTFH with settings in the compiler configuration file

isam-extfh

isam-extfh-lib

- The configuration file flags isam-extfh and isam-extfh-lib enable the usage of EXTFH drivers for Indexed ISAM files.
- Usage for Indexed files:

```
isam-extfh:<DRIVER NAME>
isam-extfh-lib:<library to use for this extfh driver>
```

flat-extfh

flat-extfh-lib

- The configuration file flags flat-extfh and flat-extfh-lib enable the usage of EXTFH drivers for Sequential/Relative Files.
- Usage for Sequential/Relative files:

```
flat-extfh:<DRIVER NAME>
flat-extfh-lib:<library to use for this extfh driver>
```

Runtime support for EXTFH

Runtime environment variables COB_EXTFH, COB_EXTFH_INDEXED, COB_EXTFH_FLAT, and COB_EXTFH_LIB allow for the detection of an EXTFH interface at runtime.

With this enhancement, you can make use of an EXTFH data source without having to compile with the –use-extfh compiler flag.

At run time define :

```
For EXTFH interface to RDBMS:
COB_EXTFH=<your extfh name>

For EXTFH interface to Indexed Files:
COB_EXTFH_INDEXED=<your extfh name>

For EXTFH interface to Sequential or Relative Files:
COB_EXTFH_FLAT=<your extfh name>

COB_EXTFH_LIB=<list of shared libs containing extfh code>
```

For example :

```
COB_EXTFH=CITEXTFH
COB_EXTFH_LIB=/opt/mytools/lib/liba.so:/opt/mytools/lib/libb.so
```

Related topics

The FCD

When using the EXTFH interface, COBOL I-O statements are all handled as CALLs to EXTFH. EXTFH implementation is publicly documented, and uses a File Control Description (FCD) structure, which is updated, and passed as a parameter in the CALL to EXTFH. COBOL-IT's implementation of the External File Handler (EXTFH) supports four file types : line-sequential, record-sequential, indexed, and relative.

The COBOL definition of the FCD is contained in the XFHFCD.CPY file, which is located included in your distribution, in the $COBOLITDIR\copy subfolder.

Accessing the FCD programmatically

The –ffcdreg compiler flag allows a user of an EXTFH compliant data source to directly read and write the File Control Description (FCD) through which information passes to and from an EXTFH-compliant data source. When the –ffcdreg compiler flag is used the compiler will generate an error if –use-extfh is not used.

As background, EXTFH makes use of a standardized File Control Description (FCD), through which information passes to and from the EXTFH-compliant data source.

An FCD is created for each file that is mapped to an EXTFH-compliant data source.
It can be useful inside a program to directly read and write the FCD. The FCDREG compiler directive was developed for this purpose, and the COBOL-IT implementation of this functionality is the –ffcdreg compiler flag. When you compile with the –ffcdreg compiler flag, a register is created for each [filename] which is named "FH--FCD of [filename]". Note that there are two hyphens in the name "FH--FCD". By describing the FCD structure, and positioning the beginning of the structure at the address of "FH--FCD of [filename]", individual elements within the structure can be

read and written.

Note- The FCD structure is described in a copy file called XFHFCD.CPY, which is included in the $COBOLITDIR\copy directory in Windows, and the $COBOLITDIR/share/config directory on UNIX/Linux-based systems.

For example:
1- Include a reference to the FCD in your Linkage Section, as follows:

```
LINKAGE SECTION.
01 FCD.
COPY "XFHFCD.CPY".
```

2- Sync the address of FCD with the address of FH--FCD OF FIL1.

```
PROCEDURE DIVISION.
. . .
        SET ADDRESS OF FCD TO ADDRESS OF FH--FCD OF FIL1.
```

3- After performing the SET statement above, the fields in XFHFCD.CPY can be read and written.

Using third-party software that requires EXTFH

COBOL-IT provides a library which can be used to access the host VBISAM file system through EXTFH, called citextfh_dll.dll (in Windows), and libcitextfh.a (in Linux/UNIX).

If a third-party library requires the External symbol EXTFH, just add to your link command the :

```
>cobc .....-lcitextfh
```

This redirects all EXTFH calls to the COBOL-IT files to the EXTFH library routines provided by COBOL-IT.

The TXSeries SFS EXTFH package- An example

For more details about accessing Structured File Server (SFS), DB2, or Oracle files with the COBOL-IT compiler through the EXTFH compatible function, see IBM documentation at :
http://www-01.ibm.com/support/knowledgecenter/SSAL2T_8.2.0/com.ibm.cics.tx.doc/tasks/t_prog_usg_exfth.html?lang=en

TXSeries SFS is a structured file server that manages access to data stored in record oriented files. SFS supports both transactional and non-transactional access to data. It supports VSAM file organizations ESDS, KSDS and RRDS.

The *External File Handler* (EXTFH) is a package that allows COBOL applications to transparently use SFS files for record storage. To the COBOL programmer there is no apparent difference between this and standard COBOL I/O; the routines to access data are the same. The only difference is that you must compile your applications with EXTFH enabled or use the runtime support for EXTFH provided by COBOL-IT.

COBOL supports four file types: line-sequential, record-sequential, indexed, and relative. When EXTFH is in use, three of these are mapped to SFS file types, as shown in Table 1.

Table 1. EXTFH file type mappings

COBOL File Type	SFS File Type
line-sequential	Not supported in SFS
record-sequential	entry-sequenced
indexed	clustered
relative	relative

Using an EXTFH-compatible file system with COBOL-IT

With the COBOL-IT compiler, you can access SFS files through the EXTFH-compatible function. An EXTFH-compatible function is supported in the COBOL-IT software and by the EXTFH code on the SFS file system managers.

cobol_Extfh is the TXSeries EXTFH interface for COBOL-IT applications.
LibEncSfsExtfhCobit is the TXSeries SFS-EXTFH library for COBOL-IT applications.

To access TXSeries SFS through the TXSeries-EXTFH library, either the COBOL-IT application must be compiled with the TXSeries-EXTFH library which is explained below in **Step 1** OR the TXSeries-EXTFH library can be detected at runtime by setting the appropriate COBOL-IT environment variables which is explained below in **Step 2**.

Step1- Compiling a COBOL-IT application with the TXSeries-EXTFH library

COBOL-IT compiler allows third-party EXTFH drivers with the compiler option **"-use-extfh" <handler_name>**

When the –use-extfh compiler flag is used, all file i-o performed using standard COBOL verbs is redirected to a call of the external symbol `<handler_name>`.

Below is the command to compile a sample COBOL-IT application test.cbl

```
cobc -x test.cbl $(CICSPATH)\lib\libEncSfsExtfhCobit.lib -use-
extfh=cobol_Extfh
```

Step 2- Runtime support for TXSeries-EXTFH:

COBOL-IT runtime can detect an EXTFH interface at runtime through the environment variables COB_EXTFH and COB_EXTFH_LIB. In this case, the application must be compiled normally without using the "-use-extfh" option.

For example:
#cobc –x sample.cbl

Export the environment variables below:
set COB_EXTFH=cobol_Extfh
set COB_EXTFH_LIB=c:\opt\cics\bin\LibEncSfsExtfhCobit.dll

Then the run the program. The TXSeries-EXTFH interface will be detected at runtime.

TXSeries SFS-EXTFH functionality for COBOL-IT on windows

Support for SFS-EXTFH functionality is provided for COBOL-IT on Windows beginning in TXSeries 7.1 fix 5. This fix also contains sample EXTFH programs which access different file types on SFS. Instructions for compiling and running the sample programs are provided in the README_extfh.txt file.

Oracle

COBOL-IT's COBOL-IT® Compiler Suite has been certified for use with Oracle Pro*COBOL, and the Oracle Database 11g Enterprise Edition, allowing users to embed SQL statements into their COBOL programs, and retrieve, manage, and manipulate corporate data stored in their Oracle database.

COBOL/ESQL operations that have been written and tested in proprietary mainframe environments do not need to be re-engineered, thereby lowering the costs and risks associated with Enterprise Application Modernization. The Oracle Pro*COBOL precompiler takes these COBOL-IT programs containing ESQL statements as input, and produces as output COBOL programs in which the ESQL statements have been translated into calls to functions in Oracle libraries. These COBOL programs can then be compiled by the COBOL-IT Compiler, with the result being object code that has access to the Oracle database.

This chapter describes the different step to needed to link Oracle(tm) with a COBOL-IT program. We suppose that Oracle Client is installed in $ORACLE_HOME and $ORACLE_HOME/bin is in your PATH.

In our example below, we examine the different steps in handling a program called testsql.cbl, which contains ESQL COBOL statements designed to interact with the Oracle database.

The first case we will examine follows the normal course of actions, which are:

* Precompile the COBOL source program with procob, Oracle's ESQL/COBOL precompiler. The process of precompiling commands out all ESQL statements and replaces them with CALLs to routines provided by Oracle, in a new output file. Topics covered include:

* Initiating the precompiler from within a script (Linux/Unix), or batch file (Windows). There may be some advantages to separating the Precompile step from the Compile step.

* Invoking the precompiler on the command line, using the –preprocess=cmd compiler flag.

* Compile the output file with cobc. Structure the compile line such that it links in the necessary libraries provided by Oracle, to ensure that the CALLs will be resolved.

Topics covered also include:

* Constructing a compiler command with link commands for Oracle libraries.

* Changes to Compiler Configuration Flags.

* Running the compiled object with cobcrun.

Then, we will examine debugging considerations. Topics covered include:

* Relinking the Deet debugger with Oracle libraries.

* How to debug original source

* How to debug precompiled source

-preprocess=cmd

To provide greater compatibility with other COBOL compilers, COBOL-IT provides the ability to invoke a precompiler on the command line, using the –preprocess=cmd compiler flag.

If your preference in debugging is to debug original source, as opposed to debugging precompiled source, you should make use of –preprocess=cmd, which provides this capability.

Note that if, while debugging original source, you need a finer level of tracing on the Exec SQL Statements in your code, you can also use the –fdebug-exec compiler flag for extra tracing capabilities.

For details on the use of -preprocess=cmd, see **Guidelines for use of –preprocess=cmd**

Precompile the COBOL source program with procob

You may have scripts which separate the precompilation step from the compilation/link steps.

In these cases, you would not need to use the –preprocess=cmd compiler flag.

First precompile the embedded SQL:

In Linux/Unix:

```
>procob iname=procobdemo.pco oname=procobdemo.cbl
```

In Windows:
Run: precomp procobdemo.pco procobdemo.cbl

```
precomp.bat

set ICHOME=C:\COBOL\INSTANTCLIENT_11_2
set PCBCFG=%ICHOME%\precomp\admin\pcbcfg.cfg
set PROCOB=%ICHOME%\sdk\procob.exe
%PROCOB% iname=%1 config=%PCBCFG% ireclen=132 oname=%2
```

Changes to Compiler Configuration Flags

Then, make changes to the Compiler Configuration File:

The Oracle Pro*COBOL runtime requires binary field to be stored on 2, 4 or 8 bytes.

The Micro Focus IBMCOMP compiler flag corresponds to the binary size setting of 2-4-8.

```
binary-size: 2-4-8
```

Another problem with Oracle Pro*COBOL runtime is the fact that Oracle provides SQLCA structures declared with fields described as USAGE COMP.

By default COMP is Big-Endian on all platforms.

On Little-Edian platform, while those fields are declared as USAGE COMP, the Pro*COBOL runtime expects "native binary fields" to be stored in Little-Endian format, which should be declared as USAGE COMP-5.

There are two possible solutions:

- You can set the binary-byteorder entry in the compiler configuration file to "native"

  ```
  # Value: 'native', 'big-endian'
  binary-byteorder: native
  ```

  Doing this causes all fields described as USAGE BINARY, USAGE COMPUTATIONAL or USAGE COMP to be stored as USAGE COMP-5, (Platform native format).

  This option is –not- recommended when you are operating on a Little-Endian platform, and using a file that has been generated on a Big-Endian platform, such as a Mainframe.

- You use the –makesyn "COMP=COMP-5" compiler flag when compiling preprocessed source. Note that when using this solution, you are making the declaration USAGE COMP synonymous with the declaration USAGE COMP-5. This usage of the –makesyn compiler flag would have no effect on data items declared as USAGE COMPUTATIONAL.

  Note that in the COBOL-IT implementation of the –makesyn compiler flag, the first word becomes a synonym of the second word.

  This is similar to the MAKESYN directive implemented by Micro Focus. The same result in Micro Focus, would be declared as : MAKESYN"COMP-5"="COMP". Note that in this imiplementation, the second word becomes a synonym of the first word.

Any changes that have been made to the compiler configuration file should be Save'd in a new configuration file called oraconf.conf. This will prevent the settings from being overwritten when you install an update to your compiler.

To have your compiler reference the new configuration file (for example, oraconf.conf), add the compiler flag

`-conf=oraconf.conf` to your compile string.

Note- You can name this configuration file whatever you want, provided it has a .conf extension, and provided that it is saved in the $COBOLITDIR/config directory (Linux/Unix) or %COBOLITDIR%\config (Windows).

A compiler command with link commands for Oracle libraries

Creating native executables

Then compile the generated source `testsql.cbl` and link Oracle libraries:

In Linux/Unix:

```
>cobc –conf=oraconf.conf -x testsql.cbl
$ORACLE_HOME/precomp/lib/cobsqlintf.o -L $ORACLE_HOME/lib/ -l
clntsh
```

NOTE this example was done on Linux SLES 10 , other platforms may require additional system library.

In Windows:

```
>set ICHOME=C:\INSTANTCLIENT_11_2
>set ICLIBHOME=%ICHOME%\sdk\lib\msvc
>set PCBCFG=%ICHOME%\precomp\admin\pcbcfg.cfg
>set SQLLIB_lib=orasql11.lib
>cobc -conf=myconf.conf -c procobdemo.cbl -o procobdemo.obj
>cobc -x procobdemo.obj %ICLIBHOME%\%SQLLIB_lib%
```

Building a new cobcrun

In Linux/Unix environments, if your preference is to use cobcrun to launch compiled objects which need to access the Oracle database, then you will need to build a new cobcrun, with commands that link the necessary Oracle libraries with cobcrun.

To build your own `cobcrun` that includes an Oracle CALL entry point:

In Linux/Unix:

```
$ cobc -x -flink-only -o cobcrun
$COBOLITDIR/lib/cobol-it/cobcrun.o
$ORACLE_HOME/precomp/lib/cobsqlintf.o -L $ORACLE_HOME/lib/
-lclntsh
```

Replace the existing cobcrun with the newly created cobcrun, and make sure it is in your PATH.

In Windows:
Note that in Windows, it is not necessary to create a new cobcrun. Windows commands for creating shared objects, and running with cobcrun:

```
>set ICHOME=C:\COBOL\INSTANTCLIENT_11_2
>set ICLIBHOME=%ICHOME%\sdk\lib\msvc
>set PCBCFG=%ICHOME%\precomp\admin\pcbcfg.cfg
>set SQLLIB_lib=orasql11.lib
>cobc -conf=oraconf.conf -c procobdemo.cbl -o procobdemo.obj
>cobc -b procobdemo.obj %ICLIBHOME%\%SQLLIB_lib%
```

Run the compiled object (native executable)

In Linux/Unix:

```
./procobdemo
```

In Windows:

```
>procobdemo.exe
```

Run the compiled object (shared object)

In Linux/Unix/Windows:

```
cobcrun procobdemo
```

In summary

You can combine the compile and run commands above into scripts (Linux/Unix) or batch files (Windows) as follows:

In Linux/Unix:

```
>procob iname=procobdemo.pco ireclen=132 oname=procobdemo.cbl
>cobc –conf=oraconf.conf -x procobdemo.cbl
$ORACLE_HOME/precomp/lib/cobsqlintf.o -L $ORACLE_HOME/lib/ -l
clntsh
>./procobdemo
```

In Windows:

```
>set ICHOME=C:\INSTANTCLIENT_11_2
>set ICLIBHOME=%ICHOME%\sdk\lib\msvc
>set PCBCFG=%ICHOME%\precomp\admin\pcbcfg.cfg
>set SQLLIB_lib=orasql11.lib
>set PROCOB=%ICHOME%\sdk\procob.exe
>%PROCOB% iname=%1 config=%PCBCFG% ireclen=132 oname=%2
>cobc -conf=myconf.conf -c procobdemo.cbl -o procobdemo.obj
>cobc -x procobdemo.obj %ICLIBHOME%\%SQLLIB_lib%
>procobdemo.exe
```

The output from procobdemo is:

```
CONNECTED TO ORACLE AS USER:  scott
```

```
SALESPERSON  SALARY      COMMISSION
-----------  ----------  ----------
ALLEN           1600.00      300.00
WARD            1250.00      500.00
MARTIN          1250.00     1400.00
TURNER          1500.00        0.00

HAVE A GOOD DAY.
```

Debugging considerations

Build a cobcdb debugger with Oracle runtime

On Linux/Unix machines, debugger access to Oracle subroutines requires that Oracle libraries be re-linked with cobcdb.

Note-To debug a COBOL executable that has been linked with Oracle libraries, you need to link the same Oracle libraries into the debugger launcher (cobcdb).To build your own cobcdb including an Oracle CALL entry point

In Linux/Unix:

```
>cobc -x -flink-only -o cobcdb
$COBOLITDIR/lib/cobol-it/cobcdb.o –lcitsupport
$ORACLE_HOME/precomp/lib/cobsqlintf.o -L $ORACLE_HOME/lib/
-lclntsh
```

Replace the existing cobcdb with the newly created cobcdb, and make sure it is in your PATH.

Using cobcrun and cobcdb with Oracle (Windows)

In Windows environments, the CALL statements generated by the precompiling process are resolved in calls to DLLs, which are provided by Oracle and installed on the Oracle client workstation. In Windows environments, it is not necessary to rebuild cobcrun, and cobcdb, as the CALL statements are resolved dynamically.

Using cobcdb with applications that make CALLs to Oracle libraries:

- In Windows environments, generate a single dynamically loadable module (DLL) that includes the SQL library (orasql11.lib for Oracle 11) provided by Oracle, as in the example below.
- The following script creates procobdemo.dll, which can then be executed with the command “cobcdb procobdemo”. This example presumes that procobdemo.pco has already been precompiled, producing procobdemo.cbl as the output file.

```
>set ICHOME=C:\INSTANTCLIENT_11_2

>set ICLIBHOME=%ICHOME%\sdk\lib\msvc

>set SQLLIB_lib=orasql11.lib

>cobc –g –conf=myconf.conf –c procobdemo.cbl –o procobdemo.obj
```

```
>cobc –b procobdemo.obj %ICLIBHOME%\%SQLLIB_lib%
>cobcdb procobdemo
```

cobcdb procobdemo using –preprocess –fdebug-exec

Using –preprocess causes the debugger to display original source, and not the translations to CALL statements produced by the precompiler.

Building a new rtsora

When using Oracle in Linux/Unix environments, you may need to rebuild rtsora.

First ensure that you have a link to cobmf (MF Command line emulator)

In Linux/Unix:

```
>ln -s $COBOLITDIR/bin/cobmf $COBOLITDIR/bin/cobc
>cd  $ORACLE_HOME/precomp/lib/
>export RTSPORTFLAGS="$COBOLITDIR/lib/cobol-it/cobcrun.c –CIT –fno-main"
>make -f ins_precomp.mk relink EXENAME=rtsora
```

This command creates the new executable in the `$ORACLE_HOME/precomp/lib` directory, and then moves it to the `$ORACLE_HOME/bin` directory. To create the new executable without moving it to the `$ORACLE_HOME/bin` directory, enter the following command:

```
>make -f ins_precomp.mk rtsora
```

About the Oracle® sample program procobdemo.pco

In order to run the Oracle® sample program procobdemo.pco, you need to download the Client software, and the Instant Client, in addition to the Oracle Database. Oracle Database and Client software needs to be installed. The Instant Client package then can be unzipped into the directory of your choice. The Oracle® precompiler procob is contained in the Instant Client package, as is the demo program procobdemo.pco, along with sample scripts for running it. We include the following observations we made about compiling and running procobdemo.

1- The sample program procobdemo.pco makes CALLs to ORASQL8.DLL. ORASQL8.DLL is located in %ORACLE_HOME%\client_1\BIN directory, as is ORASQL11.DLL, which must be substituted for ORASQL8.DLL in order to run the sample procobdemo.pco. To substitute ORASQL11.DLL for ORASQL8.DLL,

```
>ren ORASQL8.DLL  ORASQL8.DLL.BAK
>copy ORASQL11.DLL ORASQL8.DLL
```

Note that Administrator privileges are required to rename ORASQL11.DLL to ORASQL8.DLL.

2- We also observed when running the sample program, that when running procobdemo, we

initially received the error:

```
Error:  ORA-28000 - the account is locked
```

This is a well-documented issue. To resolve, we ran SQLPLUS, and ran the following command:

```
SQL>CONNECT sys/(PASSWORD) AS SYSDBA;
SQL>ALTER USER scott IDENTIFIED BY tiger ACCOUNT UNLOCK;
```

SyncSort

Syncsort's data transformation technologies improve the run-time performance of many data intensive applications through algorithm design, architecture exploitation, dynamic optimization, and constant benchmarking. It also optimizes run-time performance through state-of-the-art parallel processing technology and using the best I/O method available. This reduces CPU, memory and disk resource usage, allowing applications to be deployed on significantly smaller hardware systems, in turn lowering hardware costs considerably.

COBOL-IT's interoperability with Syncsort makes use of the MF Compliant 'External Sort Module' (EXTSM) and 'External File Handler' (EXTFH). The EXTSM interface allows COBOL-IT to swap out its internal SORT engine, used to process SORT /MERGE operations , for the SORT engine provided by Syncsort. The EXTSM interface is enabled with the –use-extsm compiler flag, and the host library routines. The EXTFH interface is used by Syncsort for the application of its highly optimized SORT/MERGE algorithms. The EXTFH interface is enabled with the –use-extfh compiler flag.

Syncsort algorithms are provided in libraries and are made available to the COBOL-IT program by linking these libraries (Unix) or, where applicable, by ensuring that the necessary DLLs are located in the host system PATH (Windows). Note- In Windows environments, the key DLL's are mfsyncsort.dll and syncsort.dll. Installation of Syncsort automatically updates the PATH with the location of these DLL's. Similarly, the LIB environment variable is updated with the location of mfsyncsort.lib.

As examples:

To compile a program with will use SyncSort as External sort Module : Supposing SyncSort is installed in $SYNCSORT_DIR:

In Linux/Unix:

```
> cobc -x -use-extsm EXTSM –lcitextfh  -L $SYNCSORT_DIR/lib     -l
mfsyncsort -l syncsort myprog.cbl
```

InWindows:

```
cobc -x -use-extsm EXTSM –l citextfh_dll.lib  -L $SYNCSORT_DIR/lib
-l mfsyncsort.lib -l syncsort.lib myprog.cbl
```

Tuxedo

This chapter describes how to use COBOL-IT with Oracle Tuxedo. For specifics about the installation and configuration of Tuxedo, please refer to Oracle documentation. Oracle Tuxedo is available on all of the platforms to which COBOL-IT is ported.

Oracle Tuxedo delivers powerful transaction monitoring technology aimed at facilitating the development and deployment of SOA applications. Oracle Tuxedo provides Client libraries, and Server-based software, and a published API that is accessible by many programming languages- and perhaps most notably the COBOL programming language. As a result, it is easy to create front-end programs in COBOL which initialize and communicate with the middleware, as it is to use the Server to initiate services written in COBOL.

In a distributed processing (client/server) environment, the interaction between COBOL-IT and Tuxedo occurs as shown in the following diagram:

COBOL-IT Front-end < > Tuxedo Client Software < > Tuxedo Server < > COBOL-IT Services

Useful references on building Tuxedo clients using the Tuxedo buildclient script can be located at http://download.oracle.com/docs/cd/E13203_01/tuxedo/tux71/html/rfcmd5.htm.

Useful references on building a Tuxedo server using the Tuxedo buildserver script can be located at http://download.oracle.com/docs/cd/E13203_01/tuxedo/tux80/atmi/rfcmd8.htm.

When transitioning from another COBOL compiler that uses the Tuxedo buildserver script, it is important to understand that cobc builds a “main”- which is the entry point for a “C” program.

To cause COBOL-IT to not build a “main”, you must use the compiler flag –fno-main.
For your purposes when integrating with Tuxedo, this can be handled by setting the environment variable COBITOPT to include the “-fno-main” setting.

See the chapter below titled : **Passing COBOL-IT compiler flags using COBITOPT** for more details.

“cobmf”- the COBOL-IT MF Command-line Emulator

cobmf, or cobmf.exe in Windows environments, is located in the $COBOLITDIR\bin directory.
For a full list of compiler flags supported by cobmf, just type cobmf [return] at the command line.

cobmf facilitates the transition from Micro Focus COBOL to COBOL-IT by providing a Micro Focus command-line emulator. The user can rename cobmf to cob (or rename cobmf.exe to cob.exe in Windows environments) , and continue to use the same compiler flags and environment variables that they have developed over time.

Due to the high level of compatibility with Micro Focus that COBOL-IT provides with its MF command line emulator “cobmf”, current documentation provided by Tuxedo for use with Micro Focus COBOL can be used by COBOL-IT users, with just a few adjustments, documented below.

Note how this applies, when referencing Tuxedo documentation, which was developed for use with the Micro Focus compiler cob, and where a number of Micro Focus environment variables are referenced. Using cobmf (renamed as cob), directs the COBOL-IT compiler to reference these, rather than the equivalent versions used by the COBOL-IT compiler cobc. Cobmf thus provides a powerful tool in transitioning, as scripts such as the one below, used to document how to set up the environment to run the Oracle Tuxedo sample with a COBOL client program, can be used "as is":

From: http://download.oracle.com/docs/cd/E12531_01/tuxedo100/tutor/tutcs.html

```
APPDIR=<pathname of your present working directory>
TUXCONFIG=$APPDIR/TUXCONFIG
COBDIR=<pathname of the COBOL compiler directory>
COBCPY=$TUXDIR/cobinclude
COBOPT="-C ANS85 -C ALIGN=8 -C NOIBMCOMP -C TRUNC=ANSI -C OSEXT=cbl"
CFLAGS="-I$TUXDIR/include"
PATH=$TUXDIR/bin:$APPDIR: $PATH
LD_LIBRARY_PATH=$COBDIR/coblib:${LD_LIBRARY_PATH}
export TUXDIR APPDIR TUXCONFIG UBBCONFIG COBDIR COBCPY
export COBOPT CFLAGS PATH LD_LIBRARY_PATH
```

In the script above, cobmf/cob recognizes COBOPT, COBCPY, and COBDIR, and reproduces the expected behaviors associated with those COBOL-oriented environment variables, while also recognizing all of the compiler flags listed in the COBOPT environment variable, and applying those, when using the COBOL compiler.

Configuring, Compiling and Linking Tuxedo programs

To configure, compile, and link Tuxedo programs:

- Rename cobmf as cob
- Ensure that cobmf/cob is in your PATH
- Use the Tuxedo-provided buildclient and buildserver scripts, adding a reference to cittuxedo.c, as described below.

Passing COBOL-IT compiler flags using COBITOPT

The tuxedo-provided buildclient or buildserver make use of a convention, in which the –C flag indicates that a COBOL compiler cob should be used, and the compiler flags should be stored in the environment variable COBOPT and COBITOPT. Note that COBITOPT is needed only for the compiler flags that are not supported by COBOPT.

To compile programs correctly for tuxedo, COBOL-IT compiler needs the followings flags:

```
export COBITOPT="-fno-main –conf=+tuxedo.symb"
```

For cases, such as the Tuxedo sample program, where command line parameters are required, the COBOL-IT compiler flag –fC-cmd-line is also required.

The version 11g of tuxedo provides a COBOL sample called CSIMPAPP.

The client part of that sample reads command line parameters with the code below:

```
 LINKAGE SECTION.
 01 OS-LEN PIC S9(9) COMP.
  02 OS-STRING.
    03 PARMPTR-TABLE OCCURS 1 TO 100 TIMES DEPENDING ON OS-LEN.
 01 PARMPTR POINTER.
 01 PARM-STRING PIC XXXXXX.
*****************************************************
*Start program with command line args
*****************************************************
 PROCEDURE DIVISION USING BY VALUE OS-LEN BY REFERENCE OS-STRING.
```

To compile this correctly, the COBOL-IT compiler requires the compiler flag -fC-cmd-line.

As a result, the script for building this sample requires the setting of COBITOPT, as follows:

In Linux/Unix

```
export COBITOPT="-fno-main –conf=+tuxedo.symb  -fC-cmd-line"
buildclient -C -o CSIMPCL  -f  CSIMPCL.cbl
export COBITOPT="-fno-main –conf=+tuxedo.symb"
```

In Windows

```
set COBITOPT="-fno-main –conf=+tuxedo.symb  -fC-cmd-line"
buildclient -C -o CSIMPCL  -f  CSIMPCL.cbl
set COBITOPT="-fno-main –conf=+tuxedo.symb"
```

Appendices

Frequently Asked Questions

What is required for deployment in Windows?

On Windows, there is not a runtime-only package. What should be distributed at Customer sites?

A. For starters, all of the files your Customer requires are in the \bin directory.
All of the DLL's in the bin directory should be distributed. They must be in the PATH when the COBOL program is run,.
If you wish to save some space, you may remove :

citmake.exe	makefile utility. Used by the Developer Studio
cobmf.exe	Translates some MF compiler commands into COBOL-IT compiler commands.

Mixing software versions creates problems

Can you install a update a development system installation with a later version of a runtime version of the software?

A. No. The runtime-only version of the software is for deployment only. It can not be used to to update an installed version of a development system. In order to perform a version update, you must uninstall/erase the previous version, and reinstall a full version of the new version. Mixing versions results in unpredictable behavior.

Compilation Fails: cannot find -lncurses

I installed the COBOL-IT Compiler Suite on a Linux Redhat platform. When I try to compile a simple COBOL program, I get the following message:
/usr/bin/ld: cannot find –lncurses
What should I do?

A. You must install the ncurses development tools. On Redhat/CentOS, the command is:
>yum install ncurses-devel

If you are on a 64-bit Linux machine, and using the 32-bit COBOL-IT compiler, you must install the 32-bit version of the ncurses development tools. On Redhat/CentOS, the command is:
>yum install ncurses-devel.i386

Unexpected behavior when two compiler versions are installed

I have two different versions of COBOL-IT installed. An older version is installed in the default directory (/opt/cobol-it). A newer version is installed in another directory (/usr/myhome/cobol-it).

I would like to switch between the two by changing the value of the COBOLITDIR environment variable.

By changing COBOLITDIR, I can invoke the newer version of the compiler installed in the my home/cobol-it directory. However the compilation fails because the newer compiler invokes libraries in the older installation, and I get a version error. What should I do?

A. This is a known limitation. Linux first looks into the default defined at compilation time. When you want to use 2 different versions of the COBOL-IT compiler on a system, the best solution is to install them in directories other than the default directory. It is best to not create the default directory in that situation.

www.cobol-it.com

May, 2018