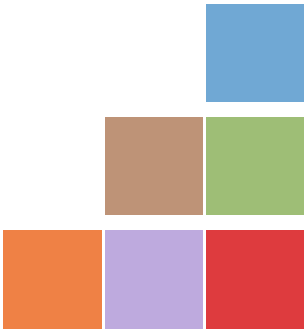
COBOL-IT® Debugger User's Guide

Version 3.11



Acknowledgment

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.

Conventions used in the General Format diagrams:

Brackets [ ] identify syntax elements that are supported but not required.

Curly Braces { } identify alternative syntax elements. Among syntax elements described within stacked curly braces, only one of the entries may be selected.

Ellipses (...) indicate (optional) repetition. If the syntax element is a required element, then it will be surrounded by curly braces.

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.

Contact Information:

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

Debugger User's Guide

Compile for debugging

The COBOL-IT Debugger is called "cobcdb". Cobcdb can be run from the command-line. Cobcdb is also integrated into the COBOL-IT Developer Studio, in the Debugger Perspective. There are many advantages to using the COBOL-IT Debugger inside the Developer Studio, as we will see in exercises, such as the Debug Attach and "C" level debugging.

For both the command-line version of cobcdb, and the Developer Studio Debugger Perspective, running a compiled object in the debugger requires that the original source code have been compiled for debugging. Compiling for debugging causes the compiler to create debugging meta data, and store it either in the compiled object itself or in a separate file.

These compile options include:

–debugdb=<DebugDB-name>

The –debugdb compiler flag causes the compiler to store debugging meta information in an SQLite3 database.

When compiling with –g, the COBOL-IT compiler stores all debugging meta information in the program binaries. This could make programs compiled for debug very huge. In some situations, it could prevent the program from loading into memory.

When compiling with the –debugdb=<DebugDB-name> compiler flag, the compiler stores debugging meta data in an SQLite3 database.

As an example, the command:
>cobc –debugdb=hello hello.cbl
creates a file called hello.dbd in addition to the compiled object.

Copy this file into the same folder as the object file. Or, if you wish to locate it elsewhere, set the COB_DEBUGDB environment variable to the full path to the DebugDB file.

The same data base should be used for the entire project.

During a debug session, the runtime debugger will check for the existence of the COB_DEBUGDB environment variable containing the full path to the DebugDB file. If the environment variable is not set, the runtime will attempt to retrieve the location of the COB_DEBUGDB data file from the compiled object.

Currently only 1 database may be used at a time. This means that the Customer must use the same one for all of his programs. Several programs may write metadata to the same database.

-g

Produce debugging information in the output.

-fdebugdb

Equivalent to `debugdb:yes` in config file

The –fdebugdb compiler flag, **when used with –g**, store alls debugging information into a file name <modulename>.dbd. **Copy this file to the same location as the the object file .so or .dll.** This will permit the runtime debugger to load the debugging information dynamically when needed.

This is different from debugdb=<filename> where you have to specify a unique Debug db for the whole project.

Compiler flags enabled by –g / -fdebug / -debugdb

-fstack-check

Enables stack checking debug function. The stack checking debug function allows the user to trace back through the stack of calling programs to the currently running line of source in a program. The –fstack-check compiler flag is enabled by the –g compiler flag, and by the –debug compiler flag.

Equivalent to `stack-check: yes` in the compiler configuration file.

-fmem-info

Enables Dump of Working-Storage when runtime aborts. The –fmem-info compiler flag functionality is enabled by the –g compiler flag, and by the –debug compiler flag.

Equivalent to `mem-info: yes` in the compiler configuration file.

-fsource-location

Generates source location code, enabling information to be dumped on source location when the runtime aborts. The –fsource-location compiler flag is enabled by the –g compiler flag, and by the –debug compiler flag.

Equivalent to `source-location: yes` in the compiler configuration file.

Compiler flags related to–g / -fdebug / -debugdb

-fdebugging-line

Enables support for debugging lines. (Source lines that contain 'D' in indicator column)
Equivalent to `debugging-line: yes`

Related:

support-debugging-line:[ok/error]

Default is support-debugging-line:ok

The support-debugging-line compiler configuration entry provides a way to override the compiler's support for usage of the character "D" in column 7 to mark a debugging line.

When set to ok (the default) , source lines that contain a "D" character in column 7 are ignored, unless the compiler configuration flag debugging-line:yes is set, in which case the line is compiled.
.
When set to error, the compiler generates an error when it encounters a "D" character in column 7.

Capturing information from a debugging session

COB_DUMP=<filename>

When a program has been compiled with –fmem-info, it stores memory information The COB_DUMP environment variable designates the filename used to dump the memory information that has been stored when a program aborts that has been compiled with the –fmem-info compiler flag.

When set to N/NO, no dump is produced. If COB_DUMP is not set, then the memory information is dumped to the file named by the COB_ERROR_FILE environment variable.

If COB_ERROR_FILE is also not set, memory information is written to stderr.

The output of this dump has been enhanced by adding the memory address of each field.
As an example:

```
WORKING-STORAGE
    RETURN-CODE [6AEF4438] = +000000000
    TALLY [6AEF4440] = +000000000
    SORT-RETURN [6AEF4448] = +000000000
    NUMBER-OF-CALL-PARAMETERS [6AEF4458] = +000000000
```

COB_ERROR_FILE=<filename>

Designates 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.

File Tracing

File tracing requires the setting of compiler flags, and the naming of a COB_ERROR_FILE to receive all runtime messages that would otherwise be sent to stderr.

-fsimple-trace

Generates trace output at runtime for executed SECTION/PARAGRAPHs.

-ftrace

Generates trace output at runtime, listing the SECTION/PARAGRAPH names as they are executed.

-ftraceall

When also compiled with –g, generates trace output at runtime, listing SECTION/PARAGRAPH/STATEMENTS names as they are executed.

Compile for runtime error-checking

-debug

Enables all run-time error checking. Runtime exception checking below for more details.

Compiler Flags enabled by –debug

The following compiler flags are enabled by use of the –debug compiler flag:

-fmem-info
- fsource-location
–fstack-check

These correspond, respectively to the compiler configuration file settings of :

mem-info: yes

source-location: yes
stack-check: yes

Runtime exception checking

Runtime exception checking is enabled when compiling with –debug, for the following compiler configuration flags. For details about the Runtime Exception Checking flags, see the file exception.def, which is located in $COBOLITDIR\include\libcob, in your distribution.

EC-ALL:[yes/no]
EC-ARGUMENT:[yes/no]
EC-ARGUMENT-FUNCTION:[yes/no]
EC-ARGUMENT-IMP:[yes/no]
EC-BOUND:[yes/no]
EC-BOUND-IMP:[yes/no]
EC-BOUND-ODO:[yes/no]
EC-BOUND-OVERFLOW:[yes/no]
EC-BOUND-PTR:[yes/no]
EC-BOUND-REF-MOD:[yes/no]
EC-BOUND-SET:[yes/no]
EC-BOUND-SUBSCRIPT:[yes/no]
EC-BOUND-TABLE-LIMIT:[yes/no]
EC-DATA:[yes/no]
EC-DATA-CONVERSION:[yes/no]
EC-DATA-IMP:[yes/no]
EC-DATA-INCOMPATIBLE:[yes/no]
EC-DATA-INFINITY:[yes/no]
EC-DATA-INTEGRITY:[yes/no]
EC-DATA-NEGATIVE-INFINITY:[yes/no]
EC-DATA-NOT_A_NUMBER:[yes/no]
EC-DATA-PTR-NULL:[yes/no]
EC-FLOW:[yes/no]
EC-FLOW-GLOBAL-EXIT:[yes/no]
EC-FLOW-GLOBAL-GOBACK:[yes/no]
EC-FLOW-IMP:[yes/no]
EC-FLOW-RELEASE:[yes/no]
EC-FLOW-REPORT:[yes/no]
EC-FLOW-RETURN:[yes/no]
EC-FLOW-SEARCH:[yes/no]
EC-FLOW-USE:[yes/no]
EC-FUNCTION:[yes/no]
EC-FUNCTION-NOT-FOUND:[yes/no]
EC-FUNCTION-PTR-INVALID:[yes/no]
EC-FUNCTION-PTR-NULL:[yes/no]
EC-I-O:[yes/no]
EC-I-O-AT-END:[yes/no]

EC-I-O-EOP:[yes/no]
EC-I-O-EOP-OVERFLOW:[yes/no]
EC-I-O-FILE-SHARING:[yes/no]
EC-I-O-IMP:[yes/no]
EC-I-O-INVALID-KEY:[yes/no]
EC-I-O-LINAGE:[yes/no]
EC-I-O-LOGIC-ERROR:[yes/no]
EC-I-O-PERMANENT-ERROR:[yes/no]
EC-I-O-RECORD-OPERATION:[yes/no]
EC-IMP:[yes/no]
EC-IMP-ACCEPT:[yes/no]
EC-IMP-DISPLAY:[yes/no]
EC-LOCALE:[yes/no]
EC-LOCALE-IMP:[yes/no]
EC-LOCALE-INCOMPATIBLE:[yes/no]
EC-LOCALE-INVALID:[yes/no]
EC-LOCALE-INVALID-PTR:[yes/no]
EC-LOCALE-MISSING:[yes/no]
EC-LOCALE-SIZE:[yes/no]
EC-OO:[yes/no]
EC-OO-CONFORMANCE:[yes/no]
EC-OO-EXCEPTION:[yes/no]
EC-OO-IMP:[yes/no]
EC-OO-METHOD:[yes/no]
EC-OO-NULL:[yes/no]
EC-OO-RESOURCE:[yes/no]
EC-OO-UNIVERSAL:[yes/no]
EC-ORDER:[yes/no]
EC-ORDER-IMP:[yes/no]
EC-ORDER-NOT-SUPPORTED:[yes/no]
EC-OVERFLOW:[yes/no]
EC-OVERFLOW-IMP:[yes/no]
EC-OVERFLOW-STRING:[yes/no]
EC-OVERFLOW-UNSTRING:[yes/no]
EC-PROGRAM:[yes/no]
EC-PROGRAM-ARG-MISMATCH:[yes/no]
EC-PROGRAM-ARG-OMITTED:[yes/no]
EC-PROGRAM-CANCEL-ACTIVE:[yes/no]
EC-PROGRAM-IMP:[yes/no]
EC-PROGRAM-NOT-FOUND:[yes/no]
EC-PROGRAM-PTR-NULL:[yes/no]
EC-PROGRAM-RECURSIVE-CALL:[yes/no]
EC-PROGRAM-RESOURCES:[yes/no]
EC-RAISING:[yes/no]
EC-RAISING-IMP:[yes/no]
EC-RAISING-NOT-SPECIFIED:[yes/no]

EC-RANGE:[yes/no]
EC-RANGE-IMP:[yes/no]
EC-RANGE-INDEX:[yes/no]
EC-RANGE-INSPECT-SIZE:[yes/no]
EC-RANGE-INVALID:[yes/no]
EC-RANGE-PERFORM-VARYING:[yes/no]
EC-RANGE-PTR:[yes/no]
EC-RANGE-SEARCH-INDEX:[yes/no]
EC-RANGE-SEARCH-NO-MATCH:[yes/no]
EC-REPORT:[yes/no]
EC-REPORT-ACTIVE:[yes/no]
EC-REPORT-COLUMN-OVERLAP:[yes/no]
EC-REPORT-FILE-MODE:[yes/no]
EC-REPORT-IMP:[yes/no]
EC-REPORT-INACTIVE:[yes/no]
EC-REPORT-LINE-OVERLAP:[yes/no]
EC-REPORT-NOT-TERMINATED:[yes/no]
EC-REPORT-PAGE-LIMIT:[yes/no]
EC-REPORT-PAGE-WIDTH:[yes/no]
EC-REPORT-SUM-SIZE:[yes/no]
EC-REPORT-VARYING:[yes/no]
EC-SCREEN:[yes/no]
EC-SCREEN-FIELD-OVERLAP:[yes/no]
EC-SCREEN-IMP:[yes/no]
EC-SCREEN-ITEM-TRUNCATED:[yes/no]
EC-SCREEN-LINE-NUMBER:[yes/no]
EC-SCREEN-STARTING-COLUMN:[yes/no]
EC-SIZE:[yes/no]
EC-SIZE-ADDRESS:[yes/no]
EC-SIZE-EXPONENTIATION:[yes/no]
EC-SIZE-IMP:[yes/no]
EC-SIZE-OVERFLOW:[yes/no]
EC-SIZE-TRUNCATION:[yes/no]
EC-SIZE-UNDERFLOW:[yes/no]
EC-SIZE-ZERO-DIVIDE:[yes/no]
EC-SORT-MERGE:[yes/no]
EC-SORT-MERGE-ACTIVE:[yes/no]
EC-SORT-MERGE-FILE-OPEN:[yes/no]
EC-SORT-MERGE-IMP:[yes/no]
EC-SORT-MERGE-RELEASE:[yes/no]
EC-SORT-MERGE-RETURN:[yes/no]
EC-SORT-MERGE-SEQUENCE:[yes/no]
EC-STORAGE:[yes/no]
EC-STORAGE-IMP:[yes/no]
EC-STORAGE-NOT-ALLOC:[yes/no]
EC-STORAGE-NOT-AVAIL:[yes/no]

EC-USER:[yes/no]
EC-VALIDATE:[yes/no]
EC-VALIDATE-CONTENT:[yes/no]
EC-VALIDATE-FORMAT:[yes/no]
EC-VALIDATE-IMP:[yes/no]
EC-VALIDATE-RELATION:[yes/no]
EC-VALIDATE-VARYING:[yes/no]
EC-XML:[yes/no]
EC-XML-CODESET:[yes/no]
EC-XML-CODESET-CONVERSION:[yes/no]
EC-XML-COUNT:[yes/no]
EC-XML-DOCUMENT-TYPE:[yes/no]
EC-XML-IMPLICIT-CLOSE:[yes/no]
EC-XML-INVALID:[yes/no]
EC-XML-NAMESPACE:[yes/no]
EC-XML-RANGE:[yes/no]
EC-XML-STACKED-OPEN:[yes/no]

Compiling with the –debug compiler configuration flag enables all of the exception checks.

When not compiling with –debug, you can enable specific exception checks by setting the associated compiler configuration flag to yes in the compiler configuration file.

Compiler flags related to exception checking

-ftrap-unhandled-exception

Equivalent to `trap-unhandled-exception: yes` in the compiler configuration file.

The –ftrap-unhandled-exception flag is useful in cases where certain EC compiler configuration file flags are set to yes, yet ON EXCEPTION/ON SIZE ERROR/ON OVERFLOW language is not present in the COBOL program. In these cases, using the –ftrap-unhandled-exception compiler flag causes the information made available to the user to be enhanced when the program aborts.

As an example, in a case where there is a compiler configuration flag setting of :
EC-SIZE:yes

and where this phrase does not contain an ON SIZE ERROR clause, the program would abort in cases where a SIZE ERROR was triggered. In combination with –ftrap-unhandled-exception:yes, all size error events will be captured.

As another example, where there is a compiler configuration flag setting of:
EC-SIZE-ZERO-DIVIDE:yes
In combination with trap-unhandled-exception:yes, setting EC-SIZE-ZERO-DIVIDE:yes will capture all division by zero error events if no ON SIZE ERROR clause is present.

Note- this applies to the following EC- compiler configuration flags:

EC-IMP-ACCEPT :yes
For Accept exception
EC-IMP-DISPLAY :yes
For Display exception
EC-SIZE : yes
#For Arithmetic exception
EC-OVERFLOW : yes
For String/Unstring exception

Note that all of the EC- compiler configuration flags can be set to yes using the –debug compiler flag. You may wish that your error procedure be always called on any exception, and thereby ensure that your server will handle it and not crash. In these cases, you should use the –debug compiler flag together with the –ftrap-unhandled-exception flag.

For details on how to install and uninstall error procedures, see the documentation for the CBL_ERROR_PROC library routine. CBL_ERROR_PROC installs or uninstalls an error procedure, which is run when a program-ending error occurs. The Error Routine allows the user to register procedures that will automatically be executed either when a program-ending error occurs.

CBL_ERROR_PROC

CBL_ERROR_PROC installs or uninstalls an error procedure, which is run when a program-ending error occurs. The Error Routine allows the user to register procedures that will automatically be executed either when a program-ending error occurs.

Usage

```
call "CBL_ERROR_PROC" using error-proc-flag,
                            error-proc-addr.
```

Parameters

- error-proc-flag
  - set to 0 to install error proc
  - set to 1 to uninstall error proc

01 ERROR-PROC-FLAG PIC X COMP-X VALUE 0.

- error-proc-addr address of error proc

01 ERROR-PROC-ADDR USAGE PROCEDURE-POINTER.

- error-proc-msg message from error

LINKAGE SECTION.

```
01 ERROR-PROC-MSG PIC X(ERROR-PROC-MSG-LEN).
```

Syntax

error-proc-flag	set to 0 to install error proc set to 1 to uninstall error proc
error-proc-addr	address of error proc
error-proc-msg	message returned through linkage

Code Sample

```
    ...
78 ERROR-PROC-MSG-LEN      VALUE 325.
01 ERROR-PROC-FLAG         PIC X COMP-X VALUE 0.
01 ERROR-PROC-ADDR         USAGE PROCEDURE-POINTER.
01 STATUS-CODE             PIC 9(4) COMP VALUE ZEROS.
    ...
LINKAGE SECTION.
01 ERROR-PROC-MSG PIC X(ERROR-PROC-MSG-LEN).
PROCEDURE DIVISION.
MAIN.
   SET ERROR-PROC-ADDR TO ENTRY "ERROR-PROC".
   CALL "CBL_ERROR_PROC" USING ERROR-PROC-FLAG,
                        ERROR-PROC-ADDR
                  RETURNING STATUS-CODE.
    ...
*
ENTRY "ERROR-PROC" USING ERROR-PROC-MSG.
   DISPLAY "IN ERROR PROCEDURE".
   DISPLAY FUNCTION TRIM(ERROR-PROC-MSG).
   DISPLAY FUNCTION EXCEPTION-LOCATION.
   EXIT PROGRAM.
   STOP RUN.
```

Controlling the verbosity of the Compiler error report

-v

Produces verbose output. The output of the –v compiler flag displays, all of the steps, and intermediate programs created by the compilation.

-err <file>

Causes errors and warnings to be written to <file> instead of stderr

-save-temps (=<dir>)

Causes all intermediate files to be preserved. Note- "intermediate files" are the "C" source and header files that are created during the compilation process. These files will be located in a subdirectory named "c", when using the –save-temps compiler flag.

Compiling and Debugging Pre-processed Code

-fdebug-exec

Affects the tracing of Exec statements when debugging code that has been compiled with the integrated pre-processor (-preprocess). When using the Integrated Preprocessor Interface, the default behavior of the debugger is to –not- trace (display) the code generated by the external preprocessor. Only the original source EXEC statements are shown. The –fdebug-exec compiler flag enables the tracing (debugging) of the generated code.

Equivalent to `debug-exec: yes` in the compiler configuration file.

-fexpand-exec-copy

The –fexpand-exec-copy compiler flag causes the compiler to expand COBOL COPY statements inside EXEC … END-EXEC blocks. This applies to both EXEC SQL and EXEC CICS blocks.

Equivalent to: `expand-exec-copy:yes` in config file

-fkeep-org-src-line

For use with the integrated pre-processor (-preprocess). Causes errors to be reported on the original source line.

Equivalent to `keep-org-src-line: yes` in the compiler configuration file.

Syntax Only Checking

-fexec-check

Used with -fsyntax-only, checks the EXEC SQL/CICS/DLI syntax

Equivalent to `exec-check: yes` in the compiler configuration file.

-fsyntax-only

Performs syntax error checking only. Output is limited to results of syntax check.

Equivalent to `syntax-only: yes` in the compiler configuration file.

-fvalidate-only

Compile source, no output produced, EXEC are ignored

Equivalent to `validate-only: yes` in the compiler configuration file.

Guidelines for using the COBOL-IT Profiler

COBOL-IT provides a profiling utility that allows you to analyze where your programs are spending time by providing output, in Excel format, on the number of times a paragraph is executed, and both CPU and elapsed time spent in each paragraph.

The COBOL-IT Profiler is enabled by using the –fprofiling compiler flag, or by setting : profiling: yes in the compiler configuration file.

-fprofiling

Generates paragraph profiling code. The output produced by the profiler includes separate Counts for CPU and real elapsed times.

The time is expressed in a platform-dependent unit, named "Ticks" as provided by the runtime environment of the "C" Compiler at hand. Please check the clock function for more information about this.

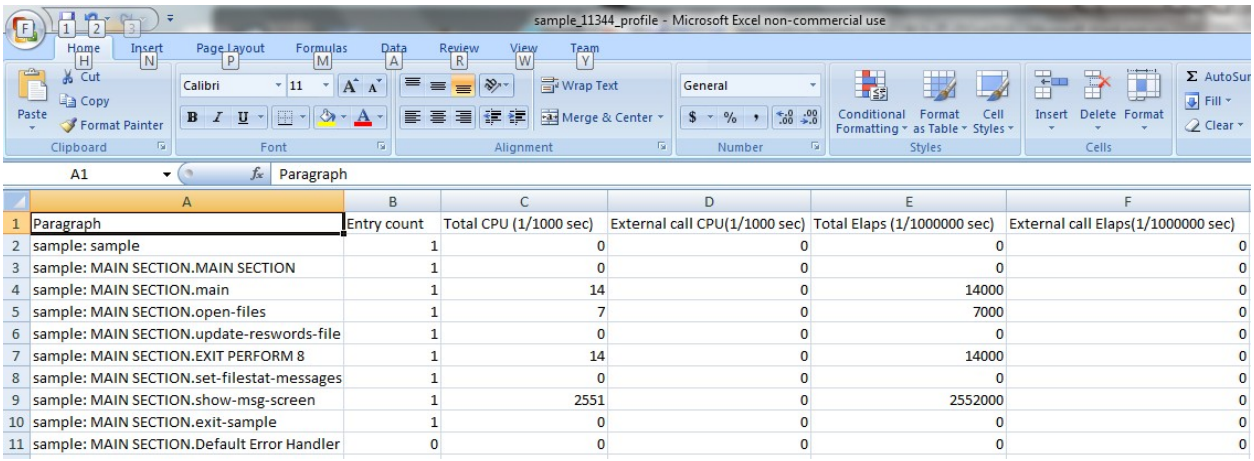
Because of the coarseness of this unit, some of the times measured as described above may be zero, while the paragraph has been executed one or more times.

On program exit, the COBOL_IT runtime generates a file named [module]_[PID]_profile.xls where [module] is the program name and [PID] is the PID number. This file is a tab separated text file, and can be opened directly with a spreadsheet like OpenOffice Calc or Microsoft Excel.

To enable the profiling utility, compile your program with the –fprofiling compiler flag.

Example:

```
>cobc -fprofiling sample.cbl
>cobcrun sample
>sample_11344_profile.xls
```



	A	B	C	D	E	F
1	Paragraph	Entry count	Total CPU (1/1000 sec)	External call CPU(1/1000 sec)	Total Elaps (1/1000000 sec)	External call Elaps(1/1000000 sec)
2	sample: sample	1	0	0	0	0
3	sample: MAIN SECTION.MAIN SECTION	1	0	0	0	0
4	sample: MAIN SECTION.main	1	14	0	14000	0
5	sample: MAIN SECTION.open-files	1	7	0	7000	0
6	sample: MAIN SECTION.update-reswords-file	1	0	0	0	0
7	sample: MAIN SECTION.EXIT PERFORM 8	1	14	0	14000	0
8	sample: MAIN SECTION.set-filestat-messages	1	0	0	0	0
9	sample: MAIN SECTION.show-msg-screen	1	2551	0	2552000	0
10	sample: MAIN SECTION.exit-sample	1	0	0	0	0
11	sample: MAIN SECTION.Default Error Handler	0	0	0	0	0

Compiler Configuration File

debug-exec: [yes/no]

Default is `debug-exec:no`.

When set to yes,
Affects the tracing of Exec statements when debugging code that has been compiled with the integrated pre-processor (-preprocess). When using the Integrated Preprocessor Interface, the default behavior of the debugger is to –not- trace (display) the code generated by the external preprocessor. Only the original source EXEC statements are shown. The –fdebug-exec compiler flag enables the tracing (debugging) of the generated code.

debugdb:[yes/no]

Default is `debugdb:no`.

When set to yes, and when used with –g, stores all debugging information into a file name <modulename>.dbd. Copy this file to the same location as the the object file .so or .dll. This will permit the runtime debugger to load the debugging information dynamically when needed.

This is different from debugdb=<filename> where you have to specify a unique Debug db for the whole project

source-location:[yes/no]

Default is `source-location:no`.

When set to yes,
Generates source location code, enabling information to be dumped on source location when the runtime aborts.

source-location:yes is enabled by the –g compiler flag and by the –debug compiler flag.

debugging-line: [yes/no]

Default is `debugging-line:no`.

When set to yes,
Enables support for debugging lines. (Source lines that contain 'D' in indicator column).

exec-check: [yes/no]

Default is `exec-check:no`.

When set to yes,
Used with -fsyntax-only, checks the EXEC SQL/CICS/DLI syntax.

keep-org-src-line:[yes/no]

Default is `keep-org-src-line:yes`.

When set to yes,
For use with the integrated pre-processor (-preprocess). Causes errors to be reported on the original source line.

mem-info: [yes/no]

Default is `mem-info:no`.

When set to yes,
Enables Dump of Working-Storage when runtime aborts.

mem-info:yes is enabled by the –g compiler flag and by the –debug compiler flag.

nostrip: [yes/no]

Default is `nostrip:no`.

When set to yes,
Causes objects and object and executable files to NOT be stripped.
Stripping an object or an executable is the action of removing system level debugging information

profiling: [yes/no]

Default is `profiling: no`.

When set to yes,
The compiler generates paragraph profiling code. The output produced by the profiler includes separate counts for CPU and real elapsed times. For more details on using COBOL-IT's built in Profiler, see Guidelines for use of Profiler below.

relaxed-syntax-check:[yes/no]

Default is `relaxed-syntax-check: yes`

Affects strictness of syntax checking rules applied by the compiler.

When set to yes,

Relaxed syntax checking rules are applied by the compiler.

simple-trace:[yes/no]

Default is `simple-trace:no`.

When set to yes,
Generates trace output at runtime for executed SECTION/PARAGRAPHs.

split-debug-mark:[yes/no]

Default is `split-debug-mark:yes.`

When set to yes,
DEBUG marks respect max 72 characters (default)

stack-check:[yes/no]

Default is `stack-check:no`.

When set to yes,
Enables stack checking debug function. The stack checking debug function allows the user to trace back through the stack of calling programs to the currently running line of source in a program.

stack-check:yes is enabled by the –g compiler flag and by the –debug compiler flag.

syntax-only:[yes/no]

Default is `syntax-only:no`.

When set to yes,
Performs syntax error checking only. Output is limited to results of syntax check.

trace:[yes/no]

Default is `trace:no`.

When set to yes,
Generates trace output at runtime, listing the SECTION/PARAGRAPH names as they are executed.

traceall:[yes/no]

Default is `traceall:no`.

When set to yes,
Generates trace output at runtime, listing SECTION/PARAGRAPH/STATEMENTS names as they are executed.

trap-unhandled-exception:[yes/no]

Default is `trap-unhandled-exception:no`.

When set to yes,
Is useful in cases where certain EC compiler configuration file flags are set to yes, yet ON EXCEPTION/ONSIZE ERROR/ON OVERFLOW language is not present in the COBOL program. In these cases, using the –ftrap-unhandled-exception compiler flag causes the information made available to the user to be enhanced when the program aborts.

For more details, see the documentation of the –ftrap-unhandled-exception compiler flag.

validate-only:[yes/no]

Default is `validate-only:no`.
When set to yes, causes the compilation of source to ignore all EXEC statements, and produce no compiled objects. Compiler errors are produced, and can be captured in an error file, (using –err, for example.

COBOL-IT runtime parameters

--debug, -d

Suspends and waits for debugger

--debug, -d --remote -r

Same as –debug but uses a separate file for events

COBOL-IT runtime environment variables

COB_DEBUG_ALLUSER=1

The COB_DEBUG_ALLUSER environment variable, when set to 1, and when defined before running a COBOL program, causes the pipes that are created by the debugger to communicate with cobcdb to have an attribute mask of 777, which provides Read/Write attributes for all users.

For the case where cob_init(. . .) has already been called, the same effect can be achieved by calling:

cob_debug_acl_alluser(rtd,1);

This will also ensure that the pipes that are created by the debugger to communicate with cobcdb have Read/Write attributes for all users.

Note- Usage of COB_DEBUG_ALLUSER, and/or COB_DEBUG_TMP may be indicated if you receive this error message opening a pipe created by the debugger:

Error opening /[path]/debug_xxx.cit for write (13: Permission denied)

COB_DEBUG_ID=<debug-id>

Defines a numeric ID that may be used to catch the program instead of the process id (PID). When defined, a debugger may attach to the program using the COB_DEBUG_ID.

Before running the program in debug, define the environment variable COB_DEBUG_ID, for example:

export COB_DEBUG_ID= <debug-id>

where <debug-id> is an integer.

Attach to the program using the debug-id:

For details on the debug attach functionality, see the documentation of the C$DEBUG library routine.

Then at run time you must define the runtime environment variable:
COB_DEBUGDB=<DebugDB-name>

When compiling with –debugDB=<DebugDB-name>, the compiler will modify the way debugging information is stored at compile-time. Instead of storing the metadata in the compiled object, the metadata will be stored in an SQLite3 database named by <DebugDB-name>.

COB_DEBUGDB=<DebugDB-name>

The COB_DEBUGDB=<DebugDB-name> runtime environment variable allows the runtime to locate this file during a debugging session, and use the debugging information. Currently only 1 database may be use at a time. As a consequence, the user must use the same database for all of the programs in his run unit.

COB_DEBUG_MODULES=<program-id1>:<program-id2>….

COB_DEBUG_MODULES is a list of program-ids, in which the entries are separated by a colon character ":". Adding the program-id of a program in your application to the list of COB_DEBUG_MODULES causes the debugger to break at the entry of that program.

This provides an alternative way to attach the debugger to a running process in cases where programs do not contain calls to "C$DEBUG", or where you do not have access to the remote attach interface in the Developer Studio.

COB_DEBUG_STARTUP_FILE=<filename>

The console debugger cobcdb can locate source files that have been re-located after compilation using the COB_DEBUG_STARTUP_FILE runtime environment variable and invoking the replace debugger command.

The COB_DEBUG_STARTUP_FILE runtime environment variable is set to the name and location of a file containing any number of commands that are executed when cobcdb is started.

export COB_DEBUG_STARTUP_FILE=<filename>

To locate a source file that has been moved, and associate it with an object compiled for debug, use the 'replace' debugger command, which changes the path to the source file.

The syntax is as follows: replace <oldprefix> : <newprefix>

The replace debugger command allows you to replace the location where the source files associated with the program being debugged are stored.

The replace debugger command replaces any prefix of the full pathname, so the command replace /dirA : /dirB will allow any program that was originally compiled in /dirA/dev/sources to have its source stored in /dirB/dev/sources.

Subsequent commands are stacked, so when typing two more commands as follows :

replace /dirC : /dirD
replace /dirE : /dirF

you will end up with a list of three possible replacements. Only the first matching

replacement will be executed.

Further usages include:

replace <no arguments>	Resets the list, removing active replacements
replace ?	Produces a list of active replacements.

Note that replace only affects the output of the list command.
The list debugger command allows you to expand the source you can see inside the console debugger as you execute your debugger commands:

```
(cobcdb)
s
-event-step
(cobcdb)
-event-end-stepping-range #0 CUSTOMER0 () at /opt/cobol-it-
64/samples/customer0.cbl!99
.0000099>           CALL "C$PID" USING PID.
list
.0000094.
.0000095.
*****************************************************************
.0000096.       PROCEDURE DIVISION.
.0000097.
.0000098.       Main Section.
.0000099>           CALL "C$PID" USING PID.
.0000100.          DISPLAY "PID = " PID.
.0000101.      *    CALL "C$DEBUG"
.0000102.          ACCEPT W-SYS-DATE FROM DATE.
.0000103.          MOVE W-SYS-YY                TO  CURR-YY.
.0000104.          MOVE W-SYS-MM                 TO  CURR-MM.
(cobcdb)
```

Other commands such as info sources or break will still produce the original pathname as it was stored in the binary code of the program. Other commands such as break require a match with the original pathname in order to be executed.

COB_DEBUG_TMP=<directory>

Default is /tmp.
COB_DEBUG_TMP control where the files and pipes created by the debugger are stored.

The runtime debugger uses named pipes to communicate. These are pipes with a file name, and by default, they are located in /tmp. You may relocate them by defining the COB_DEBUG_TMP environment variable.

This variable can be set in the login script of the user used to connect the remote debugger, as defined in the remote connection tab of the Developer Studio This variable can also be set in the local runtime environment. It does not need to be set in both locations. If it is set in both locations, the settings should be identical, or the settings will be ignored, and the default value of /tmp will be

used.

The COB_DEBUG_TMP environment variable may be required when debugging remotely attaching to a running process using the Developer Studio Remote System Explorer. This could be the case if COBOL-IT user:group that the program is running under have different permissions on pipe files created by default in the /tmp directory than the user:group of the user running the debugger. This problem is resolved by use of the COB_DEBUG_TMP environment variable which can be used to relocate the named pipes used by the runtime debugger into a directory in which the permissions of the user:group running the program and the permissions of the user:group running the debugger are the same.

Note- Usage of COB_DEBUG_ALLUSER, and/or COB_DEBUG_TMP may be indicated if you receive this error message opening a pipe created by the debugger:

Error opening /[path]/debug_xxx.cit for write (13: Permission denied)

COB_FILE_TRACE=[Y/N]

Default is N

When set to Y, 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. The COB_FILE_TRACE runtime environment variable is evaluated when the OPEN statement is executed by the runtime. Changes to the COB_FILE_TRACE runtime environment variable can be made during the runtime session.

COB_NO_SIGNAL=[Y/N]

Default is N.

When set to Y, causes the runtime to not catch the signal which lets the system build a core dump. Setting COB_NO_SIGNAL can improve performance, while reducing the diagnostic capabilities of the runtime.

COBOL-IT Library Routines

C$DEBUG

C$DEBUG is a library routine which can be called using either the PID of the runtime session, or the value of the environment variable COB_DEBUG_ID. Prior to calling C$DEBUG, the program should acquire the value of the PID / COB_DEBUG_ID.

You may acquire the value of the PID of the runtime session by calling the C$PID library routine, using a PIC 9(n) parameter. The parameter must be numeric, and large enough to hold the value of

the Process ID.

For example :
77 ws-pid PIC 9(5).
…..
CALL « C$PID » USING ws-pid.
CALL « C$DEBUG » USING ws-pid.

You may also call C$DEBUG USING the value of the runtime environment variable COB_DEBUG_ID. Using the runtime environment variable COB_DEBUG_ID to hold the value of this parameter has an advantage if you prefer to set the value of the parameter yourself. Acquire the value of COB_DEBUG_ID programmatically before calling the C$DEBUG library routine. The parameter must be numeric, and large enough to hold the value of the value of the runtime environment variable COB_DEBUG_ID.

For example :
77 ws-did PIC 9(5).
…..
ACCEPT ws-did FROM ENVIRONMENT « COB_DEBUG_ID ».
CALL «C$DEBUG » USING ws-did.

After a call to C$DEBUG is made, the executing program, or subprogram is paused. In this state, the COBOL-IT Debugger may be attached to this runtime process from the COBOL-IT Developer Studio.

CBL_DEBUGBREAK

CBL_DEBUGBREAK is a synonym for C$DEBUG. CBL_DEBUGBREAK is a library routine which can be called using either the PID of the runtime session, or the value of the environment variable COB_DEBUG_ID.

For example :
77 ws-pid PIC 9(5).
…..
CALL « C$PID » USING ws-pid.
CALL « CBL_DEBUGBREAK » USING ws-pid.

For more details, see the documentation of the C$DEBUG library routine.

C$PID

C$PID retrieves the Process ID of the current process.
Note that C$PID is not currently available on Windows platforms.

Usage

```
CALL "C$PID"  USING process-id.
```

Parameters

process-id PIC 9(n).

Syntax

process-id is a numeric data item which must be large enough to hold the process-id.

Code Sample

```
      *
       77 PROCESS-ID   PIC 9(7).
       ...
           CALL "C$PID" USING PROCESS-ID.
           ...
      *
```

Key concepts

- In order to attach to the COBOL-IT Debugger, the program containing the call to C$DEBUG library routine must be compiled with –g.
- The COBOL-IT Developer Studio will request the location of the source file associated with the program/subprogram that has been paused by the C$DEBUG command, for purposes of debugging.
- The COBOL-IT Developer Studio attaching to the paused runtime session requires a COBOL Project, and requires that some configuration. Recommended settings are :
    - Window>Preferences>Run/Debug>Perspectives>Open the associated perspective when lauching (Always)

The COBOL-IT Debugger Engine (cobcdb)

The COBOL-IT Debugger Engine (cobcdb) has been designed to operate as an engine, working in the background, behind a user interface, such as the interface that is provided by the COBOL-IT Debugging Perspective in the Developer Studio. The COBOL-IT Debugger Engine (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 Engine, 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:)
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 rk_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:
```

-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* 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 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

Debugger Commands include:

break

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)

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)

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 <x>

causes breakpoint number *x* to be deleted.
Example:
(cobcdb)
delete 3
^done
(cobcdb)

frame <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

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:
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 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 profiling

Causes a profiling dump to be produced, dumping profiling information at the current point in the program. Profiling information is displayed, and then dumped in the .xls file format.

Example:

(cobcdb)
info profiling

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

The list debugger command requires that the source file be accessible. The list debugger command allows you to expand the source you can see inside the console debugger as you execute your debugger commands:

```
(cobcdb)
s
-event-step
(cobcdb)
-event-end-stepping-range #0 CUSTOMER0 () at /opt/cobol-it-
64/samples/customer0.cbl!99
.0000099>           CALL "C$PID" USING PID.
list
.0000094.
.0000095.
*****************************************************************
.0000096.       PROCEDURE DIVISION.
.0000097.
.0000098.       Main Section.
.0000099>           CALL "C$PID" USING PID.
.0000100.           DISPLAY "PID = " PID.
.0000101.       *    CALL "C$DEBUG"
.0000102.           ACCEPT W-SYS-DATE FROM DATE.
.0000103.           MOVE W-SYS-YY                TO  CURR-YY.
.0000104.           MOVE W-SYS-MM                TO  CURR-MM.
(cobcdb)
```

next

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

causes an exit from the debugger.
Example:

```
(cobcdb)
quit

C:\COBOL\COBOLIT\samples>
```

replace

To locate a source file that has been moved, and associate it with an object compiled for debug, use the 'replace' debugger command, which changes the path to the source file.

The syntax is as follows: replace <oldprefix> : <newprefix>

The replace debugger command allows you to replace the location where the source files associated with the program being debugged are stored.

The replace debugger command replaces any prefix of the full pathname, so the command replace /dirA : /dirB will allow any program that was originally compiled in /dirA/dev/sources to have its source stored in /dirB/dev/sources.

Subsequent commands are stacked, so when typing two more commands as follows :

replace /dirC : /dirD
replace /dirE : /dirF

you will end up with a list of three possible replacements. Only the first matching replacement will be executed.

Further usages include:

replace <no arguments>	Resets the list, removing active replacements
replace ?	Produces a list of active replacements.

Note that replace only affects the output of the list command.
The list debugger command allows you to expand the source you can see inside the console debugger as you execute your debugger commands:

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

step

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 -[n]

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

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

Returned when a breakpoint is hit.

-event-continue

Returned by the continue command. Terminated by –event-breakpoint-hit.

-event-contreturn

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

-event-end-stepping-range

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

-event-next

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

-event-program-exited

Returned by the kill command.

-event-step

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: >cobcdb 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.
```

Check vs all of these

Debug-oriented compiler flags have performance penalties. When your code is well-tested, these compiler flags may no longer be needed, and can be removed to achieve better performance.

-debug	Turns on exception checking
-debugdb=<debugDB>	Stores metadata for debugging in SQLite3 database.
-fdebug-exec	Used for debugging of EXEC SQL statements.
Exception-checking (EC-xxx) compiler configuration flags. As an example: EC-SIZE:yes	Enabled with –debug
-fmem-info	Stores memory information, for analysis in the eventual cause of a crash.
-fprofiling	Adds counters to total statistics for reports on where your application is spending the most time.
-fsource-location	Generates source location code, enabling information to be dumped on source location when runtime aborts. Enabled by –g.
-fstack-check	Enables stack checking debug function.
-ftrace -fsimpletrace -ftraceall	The tracing compiler flags. Cause output to be written to an output file during the runtime execution.
-ftrap-unhandled-exception	Provides additional information when runtime aborts.
-g	Causes debugger metadata to be stored in the compiled object file or, if –DebugDB compiler flag is used, in an SQLite3 database.
-G	Produces debugging information, for purposes of debugging programs written in "C".
COB_ERROR_FILE	Used when debugging are set to capture information for debugging purposes.
COB_FILE_TRACE	Causes data to be written to the COB_ERROR_FILE whenever there is a file I/O operation executed.
COB_DUMP	No longer required after your functionality tests have been

	completed. Creates an output file for the memory dump created when a runtime aborts.

www.cobol-it.com

May, 2018