Method to display the level and serviceability strings of your application

Angel Rivera, Rick Russell

TR 29.3449

VisualAge TeamConnection Technical Support Team
IBM Software Solutions
Research Triangle Park, North Carolina, USA

Finding out the level of an application

ABSTRACT

This technical report provides a method that developers can use to allow their customers to find out the level (version, release, modification) of their application and related information from embedded serviceability strings gathered during the compilation of the application, such as what level or driver was used, which version and release, which operating system, etc.

There are two aspects to the method aspects:

- A set of serviceability strings that use the special @(#) string identifiers that can be extracted from executable or shared library files by the "what" utility. This method has been extremely useful with VisualAge TeamConnection and CMVC.
- A stand alone tool that has the sole purpose to provide all the serviceability information related to the level of the application.

If all the appropriate serviceability strings are used, then this method will significantly reduce the time to find out the actual version of the product that the customer is running. Furthermore, it will reduce the diagnostic time for catching mistakes when the wrong file sets are installed (problems that otherwise could take a very long and frustrating time to resolve).

ITIRC KEYWORDS

- Level of an application
- Serviceability

ABOUT THE AUTHORS

Angel Rivera

Mr. Rivera is an advisory software engineer and is the serviceability architect and RAS focal point for IBM e-Business Tools.

He joined IBM in 1989 and since then had worked in the development and support of library systems. He was the technical lead for CMVC Version 2 for many years. He was also the team lead of the technical support team for VisualAge TeamConnection.

Mr. Rivera has an M.S. in Electrical Engineering from The University of Texas at Austin, and a B.S. in Electronic Systems Engineering from the Instituto Tecnologico y de Estudios Superiores de Monterrey, Mexico.

Rick Russell

Mr. Russell is a software engineer with the VisualAge TeamConnection development and technical support team. He joined IBM in 1998 and since then has worked in the development and support of library systems.

Mr. Russell has an M.S. in Computer Science and Engineering from Penn State University.

He currently provides technical support for customers of VisualAge TeamConnection.

Introduction

This technical report provides a method that developers can use to allow their customers to find out the level (version, release, modification) of their application and related information from embedded serviceability strings gathered during the compilation of the application, such as what level or driver of the source code was used, which version and release, which operating system, etc.

If all the appropriate serviceability strings are used, then this method will significantly reduce the time to find out the actual version of the product that the customer is running. Furthermore, it will reduce the diagnostic time for catching mistakes when the wrong file sets are installed (problems that otherwise could take a very long and frustrating time to resolve).

This document will provide a concrete example of the methodology. We believe that providing concrete examples that developers could use, would facilitate its usage:

- A sample application called "myApp" is shown in its original form, which does not use the methodology explained in this document. The source code is very simple because we are not concerned with other aspects of the development of an application. Also, the original Korn shell scripts and Windows command batch files are shown. This provides a baseline that will give a reader a point of reference. We are providing a set of C source code and another set in Java.
- The source code will be modified to contain the searchable strings mentioned in this document; correspondingly, the associated scripts will also be modified to reflect the new concepts. The objective is to show the reader, the relative little "delta" effort that is needed in order to implement the methodology.
- Also, the new standalone executable "myAppLevel" will be created which will be used to indicate the version information as well as other important serviceability information. This executable should be delivered with the application and should be present in subsequent fixpaks or patches. This tool is useful for applications that are standalone or that have few components and which are delivered at the same time. However, it does not work in component suites in which each component is delivered by itself.

The structure of this document is the following:

- Chapter "Using the proposed methods to find out the level of an application in the field" explains the methodology:
  - The description of a set of serviceability strings that can be extracted via the "what" utility.
  - The description of a standalone tool that displays the important serviceability information.
- Chapter "What are the problems addressed with this method" describes the problems that this methodology intends to avoid.
- Chapter "How to embed the serviceability strings in C language" describes how to embed the serviceability strings in the executables, using the C language.

• Chapter “How to implement the standalone tool in C language” describes how to implement the standalone tool, using the C language.
• Chapter “How to get the files mentioned in this document” explains how to get the source code, Korn shell scripts and Windows command batch files mentioned in this document.
• Chapter “How to embed the serviceability strings in Java” describes how to embed the serviceability strings in the executables, using Java.

Why not simply use expandable keywords from TeamConnection or CMVC?

Certain software configuration management (SCM) tools, such as VisualAge TeamConnection and CMVC, support expandable keywords which are included in text files. However, there is a limited number of strings and there are no strings for operating system information. That is, an approach that only uses expandable keywords will not cover the whole spectrum of usages covered by the method described in this document.

Therefore, because we want to keep the proposed method as simple as possible and because expandable keywords are not sufficient, their use is not exploited here.

Was is the relationship of this method with RAS?

The acronym “RAS” means “Reliability, Availability and Serviceability”. The technique that we propose is part of the “S” in RAS: serviceability.

The technique of embedding searchable strings in the executable code is not new. The Unix "what" command is used for this purpose and some vendors use it (such as Sun), some barely use it (such as IBM) and some do not use it. We would like to motivate more IBM developers to consider the use of these strings, and that is one of the objectives for writing this document.

Also, we take a step beyond the simple act of embedding strings by providing a technique of preparing a stand alone tool that avoids the end user to type "geeky" commands such as the "what" and "grep"

Are there translation issues involved?

In Unix applications, one of the main problems during installation is that the message catalog and the locale setting might be incorrect for an application and this may cause the newly application (including the proposed standalone level tool) to fail to execute. Thus, we cannot rely on the assumption that the end user has the correct settings for the message catalog and locale. If the strings were translated and placed in a message catalog, the end user may not be able to get any translated messages from the standalone tool. Therefore, the sample source code shown in this document actually embeds the English text strings inside the samples.

Disclaimer

This technical report is not an official publication from the VisualAge TeamConnection support group. The authors are solely responsible for its contents.

Acknowledgments

We would like to acknowledge the contribution of the following co-workers:

- Lee Perlov, IBM WebSphere team, who provided suggestions on using the “what” utility.
- Tim Orlowski, IBM VisualAge TeamConnection team, who developed scripts to handle the “build time” serviceability strings.
- Werner Hahn, IBM Austin, who developed the workaround to avoid the removal of static strings by optimization phase of the C compiler.
- Steve Brock, who pioneered several years ago in our support team the usage of the “what” utility for serviceability purposes.

How to get the most up to date version of this technical report

The most up to date version of this technical report can be obtained from the following IBM VisualAge TeamConnection Enterprise Server web site:

ftp://ftp.software.ibm.com/ps/products/teamconnection/papers/trlevel.pdf

Using the proposed methods to find out the level of an application in the field

This chapter provides the usage information of the proposed methods to find out the level of an application in the field.

A set of embedded serviceability strings to be extracted by the "what" utility

Note for Intel users: The "what" and "grep" utilities described in this section are standard utilities provided by Unix. Windows users can search the Internet for similar applications. See the appendix "How to get the tools" at the end of this document.

The method of using embedded serviceability strings that are extracted by the "what" utility has been extremely useful with VisualAge TeamConnection and CMVC, because we can easily tell which is the actual version of the product that the customer is running.

The developers of an application can embed a set of serviceability strings that use the special @(#) string identifiers and can be extracted from executable or shared library files by the "what" utility.

The "what" utility searches a file and displays the strings that have the substring @(#). This combination of 4 characters does not appear in normal situations. Thus, they were chosen to be the special combination of searchable keywords by the "what" utility.

In a situation when a customer contacts the technical support personnel, the support person can tell the customer to execute the "what" utility against one of the executable files provided by the product in order to show the embedded strings. For example, let's assume that the binary file "myApp" is the executable file for an application; thus, to find out the serviceability strings embedded in this file, the customer could perform the following command:

```
what myApp
```

And the resulting output would be displayed:

```
myApp:
         myApp Code generated in Platform: AIX oem-ppc3 2 4 000022559000
         myApp Code generated using DBMS: DB2_UDB_5.2_Fixpak_11
         myApp Release: 1.2.3.4
         myApp Driver: 2001-01-16
         myApp IBM My Application, (C) Copyright, IBM Corp.,1999,2001
```

Because some operating systems (such as Solaris) use a lot these searchable keys in their include libraries. This means that if you just simply use in Solaris the "what" command against your application that has your serviceability strings, you may get a very long list of items and you may need to spend some time to find out your strings. Thus, to easily find your strings, it is

recommended to use an additional token related to the application to identify those strings that are added to the product by the developers. For example, the searchable string "myApp" is used in the example provided in this document, and in this way it is possible to use the "grep myApp" command after issuing the "what" command and quickly get only the desired strings. By the way, the "grep" utility is a standard utility provided by Unix. It searches a stream of text for a searchable keyword.

In case the "what" command displays other strings (such as from include libraries from Solaris), the customer could specify a more detailed command which will show ONLY the desired strings that have the sub-string "myApp":

```
what myApp | grep myApp
```

Standalone tool that displays the important serviceability information

This method has been extremely useful with VisualAge TeamConnection: the tool is called "tclevel". This tool was modeled after the "db2level" utility introduced in DB2 UDB 5.2.

The usage of the "what" and "grep" utilities to get the serviceability information for an application is rather "low-level" or "too techie", mostly suited for advanced technical customers; furthermore, these utilities are not readily available in Windows platforms. Novice customers (and Windows customers) would appreciate avoiding those utilities and executing something easier.

The developers of the application can provide a small standalone executable tool (which is shipped with the product update) that has the sole purpose of providing all the serviceability information related to the level of the application, such as:

- Copyright information, including year and company name.
- Target operating system name and version, if applicable.
- Target database management system (DBMS), if applicable.
- Complete version-release-modification (VRM).
- Fixpak or temporary fix information (if applicable).
- Additional data used during the compilation of the application, such as:
  - Level or driver name.
- Indicators if the code was compiled with special parameters, such as:
  - If the executable was compiled with DEBUG options.
  - If the executable was compiled as a BETA version.

This tool can be easily run by customers. For example, the customer could go to the command prompt window and enter:

```
myAppLevel
```

And the resulting output will be displayed:

```
@(#) myApp IBM My Application, (C) Copyright, IBM Corp.,1999,2001
@(#) myApp Code generated in Platform: Windows NT Version 4.0
@(#) myApp Code generated using DBMS: DB2_UDB_5.2_Fixpak_11
@(#) myApp Release: 1.2.3.4
@(#) myApp Driver: 2001-01-16
@(#) myApp *** This is BETA code ***
@(#) myApp Patch:  1
```

Other advantages of this tool are:

- It is a totally stand-alone application. It does not use shared libraries or message catalogs. This is VERY useful in situations in which there is an installation or configuration problem. If shared libraries or message catalogs were needed by the tool and there were this type of problem, then the tool might not run.
- If the product is supported in multiple platforms, then the customer support representatives will have a common way to find out the level of the application in all those platforms. That is, they will not need to remember which commands are needed for EVERY single platform.

What are the problems addressed with this method

When a product is deployed in the field, it is important for all people involved (customer, support personnel, developers) to know the exact version of the product that the customer is running. This is specially important when fixes to the product have been released; it is critical to know the precise version of the product.

Definition of product updates

These product updates in the field can take place in different ways, such as:

* A new version (such as version 7).
* A release (such as release 7.2).
* A new fixpak or service pak or modification (such as 7.2.3).
* A new level or driver (such as 7.2.3.2).
* A “hot” patch or “hot” fix or temporary fix (such as 7.2.3.2 hotfix 1).

Traditional problems

This section describes some of the problems in finding out the version of the product that occur with the most common approaches.

There is no universal standard to install a product and to find out its version

Different operating systems have different methods of installing applications and by the same token, different ways of finding out the version of the installed products. For example:

* In Windows NT the applications are usually installed via InstallShield and via Windows Installer in Windows 2000, but it is up to the packaging utility to update the registry with the proper information. The user may need to access the registry directly (with the specific key) to find out the version information of the application.

* In AIX, applications are usually installed via the “smit” tool using the installp format (IBM proprietary format). The command “lslpp” can be used to obtain the version information.

* In HP-UX, applications are usually installed via the HP-UX proprietary format. The command “swlist” can be used to obtain the version information.

* In Solaris, applications are usually installed via the Sun proprietary format. The command “pkginfo” can be used to obtain the version information.

Some applications bypass the normal installation procedure for an operating system

Some applications are installed by means of zip files or tar files, which bypass the normal installation procedures without updating the registry or the product database for the operating system. This delivery method has the following disadvantages:

* The operating system does NOT have any formal information about the product that is installed. Thus, the customer cannot use the normal administration commands to find out the version of the application.

* The uninstallation of the product might be cumbersome, because the normal uninstallation procedures from the operating system are not used.

However, this method has the advantage (for some customers) that the location of the directory of the application can be specified by the customer.

Installation of the wrong fileset

This section is intended for those products that are built for multiple operating systems or multiple database management systems (DBMSs):

* The customer may have unpackaged and installed the wrong installation file for the target operating system. For example, the customer unpackaged in AIX the tar file for HP-UX, and because the binaries are not compatible, the application will fail to execute and the support team and the customer might start chasing a problem that does not exist in the code. If the customer searched the serviceability strings, the version information very likely will appear to be correct, but will not give an indication that it is for the wrong operating system.

This means that besides the version information, it is important to add also what is the target operating system for the executable file.

* The customer may have unpackaged the wrong installation file for the target database management system (DBMS). For example, the customer unpackaged the tar file for the server that works on DB2 UDB, but the customer has installed only Oracle. This situation is similar to the one above for the wrong operating system.

This means that besides the version information and the target operating system information, it is important to add also what is the target. This is important if the application is built for different DBMSs.

* The above issues could be extrapolated to other important pre-requisite or co-requisites. That is, besides the information about the version, target operating system and DBMS, other serviceability information should be readily available.

Potential mismatch between client (GUI) and server components

Some applications that have a GUI component place the version information under the “About the product” option in the Help menu. The problem is that this information may only be directly relevant to the client GUI. That is, when the customer looks at the version of the client, the customer may assume that it represents the version of the server. If the server code is updated to a newer version, that version will also generally not be seen in the “About the product” dialog on the client.

Hot or temporary fixes might be not be properly registered with the operating system

When updates to the code are installed, these updates may not update the registry with the necessary information, especially for fixpaks or “hot” or ”temporary” fixes. Thus, it is important to find out explicitly the version information of these updates.

How to embed the serviceability strings in C language

This section describes how to embed the serviceability strings in the executable file for a sample application called “myApp” which is developed in the C language. This application could be built using make files in multiple platforms using multiple DBMSs; specifically, the cases for AIX and Windows NT are thoroughly explained.

The files mentioned in this section are available from the “c” subdirectory of the expanded zip file referred to in the section “How to get the files mentioned in this document “.

The structure of this chapter is as follows:

- Part 1: Original structure of the files.
  A sample application called “myApp” is shown in its original form, which does not use the methodology explained in this document. The source code is very simple because we are not concerned with other aspects of the development of an application. Also, the original Korn shell scripts and Windows command batch files are shown. This provides a baseline that will give a reader a point of reference.
- Part 2: Modifications to include the serviceability strings
  The source code will be modified to contain the searchable strings mentioned in this document. The objective is to show the reader, the relative little “delta” effort that is needed in order to implement the methodology.

Part 1: Original structure of the files

Let’s start with the description of the simple application myApp which is written in C and which does not contain the serviceability strings. The idea is to provide a baseline to help understand the modifications explained later on.

* The overall picture of the source code is the following:

```
myApp.c (main)
  |
  +---> Includes stdio
```

* The overall picture of the processing is the following:

```
do-build
  |
  +  Preparation step: set flags, deletes temp files
  |
  +  Invokes the make file with appropriate flags
                  |
                  +  Invokes C compiler
```

Original source code for myApp.c

Notice that this is a very small and simple program.

```
/*
NAME:    myApp.c

PURPOSE:
   This file provides the main() for the sample application.

NOTES:
   To compile:
     Unix:      cc myApp.c -o myApp
     Windows:  icc myApp.c /Fe myApp.exe
*/

#include <stdio.h>

int main(void)
{
   printf("The myApp executable is running.\n");
   printf("It prints this message and then exits.\n");
   return 0;
}

/* end of file */
```

Original source code for the make file

Unix makefile: myApp-x.mak

```
# NAME: myApp-x.mak
#
# PURPOSE:
#   This is the makefile for building the myApp application in Unix.
#
# NOTES:
# * To build the application:
#      make -f myApp-x.mak myApp
#

COMPILER=cc
CFLAGS=

all:  myApp

myApp: main.o
      $(COMPILER) -o myApp myApp.o

myApp.o: myApp.c
      $(COMPILER) $(FLAGS) -c myApp.c

# end of file
```

Windows makefile: myApp-w.mak

```
# NAME: myApp-w.mak
#
# PURPOSE:
#   This is the makefile for building the myApp application in Windows.
#
# NOTES:
# * To build the application using Windows 32-bit (VisualAge C++):
#      nmake -f myApp-w.mak myApp.exe
#

COMPILER=icc

all:  myApp.exe

myApp.exe: myApp.obj
      $(COMPILER) /Fe myApp.exe myApp.obj

myApp.obj: myApp.c
      $(COMPILER) $(FLAGS) /C+ myApp.c

# end of file
```

Original source code for the scripts to invoke the make file

Korn shell script: do-build.ksh

```
#!/usr/bin/ksh
#
# NAME: do-build.ksh [ debug | beta ]
#
# SAMPLE INVOCATION:
#   do-build.ksh debug
#
# PURPOSE:
#  This is the top level script to build the application "myApp".
#

# Remove files

rm myApp
rm *.o

# Process option

if [ $# -ge 1 ]
then
  typeset -u OPTION=$1
  echo "option=$OPTION"
  case "$OPTION" in
    BETA|beta)
      FLAGS='-DBETA'
      ;;
    DEBUG|debug)
      FLAGS='-DDEBUG -g'
      ;;
    *)
      print "Unrecognized option: $OPTION"
      ;;
  esac
fi

# Invoke the make file:

make -f myApp-x.mak all FLAGS="$FLAGS"

# end of file
```

Rexx command file: do-build.cmd

```
/* */
/*********************************************************************

NAME: do-build.cmd [ debug | beta ]

PURPOSE:
 This is the top level script to build the application "myApp".

SAMPLE INVOCATION:
  rexx do-build.cmd debug

*********************************************************************/


/* Remove files */


'@del *.obj'
'@del *.exe'

/* Verify the input parameters */

parse arg option
usage = "do-build.cmd [ debug | beta ]"

/* Do the processing only if all variables are specified */

if (option = "") then
do
  FLAGS=""
end
else
do
  if (option = "debug") then
  do
     FLAGS="/Ti+ /DDEBUG"
  end
  else
  do
     if (option = "beta") then
     do
        FLAGS="/DBETA"
     end
     else
     do
       say "*** Error: wrong option. If specified, it must be: debug or beta"
       say usage
       exit 1
     end
  end
end


say "do-build.cmd: begin"

/* Invoke the make file */

say 'nmake -f myApp-w.mak all FLAGS='FLAGS'
    'nmake -f myApp-w.mak all FLAGS="'FLAGS'"
/* end of file */
```

Part 2: Modifications to include the serviceability strings

There are 2 types of serviceability strings, from the point of view of their creation times. All these strings will eventually be placed in the source code include what-info.c and the header include what-info.h:

- Strings that seldom change, such as the copyright information, the product name, the company name. For this reason, in this document these strings are referred as “static strings”. They are defined in the “level.list” file and handled inside the script “do-what”. This file is the one that can be created, checked-out, modified and then checked-in from a library system, only when of these variables needs to be changed (not a daily event). We felt that was too cumbersome to add into this file those variables that may change daily, because we did not want to have the overhead of checking out and checking in every day this file this file.
- Strings that change frequently or that can be obtained from the environment or have additional processing. It is not worthy to add them into the level.list file because it would cause a lot of overhead with the software configuration system. Instead, these variables are handled by the main build script “do-build” and then added to a temporary “master.list” file, which in turn is handled by the script “do-what”. A sub-classification of these variables is shown below:
    - Strings that change frequently and which are specified by the user, usually when a build activity is performed, such as the driver/level. These strings are referred as “build-time strings”. They are passed as parameters or as environment variables to the script “do-build”.
    - Strings that can be obtained from the environment, such as the version of the operating system used during the build. These strings are referred as “environmental strings”.
    - Strings that do not fit neatly into the above categories, such as when building beta code or debug code. These strings are referred as “miscellaneous strings”.

The main modifications to include the serviceability strings are:

* A permanent file, named “level.list”, that has a set of variable-value pairs which specify the “static” searchable strings to be embedded in the executables. This file can be stored in a software configuration library. These are the strings that do not change very much. The format is one row per entry; each entry is composed of a variable name, a space character and a variable value which is the rest of the row. For example:

```
  productInfo IBM My Application
copyrightInfo (C) Copyright, IBM Corp.,1999,2001
  releaseInfo Release: 1.2.3.4
     dbmsInfo DBMS:    DB2 UDB 5.2 Fixpak 11
    patchInfo Patch:   0
```

* A temporary file, named “master.list”, which will be recreated during each build and which will have the strings from the “level.list” file and will have additional strings (this is done inside do-build). Eventually this master.list will be handled by the script “do-what” to generate the include files “what-info”, which are described next. For example, the whole contents of the master.list file includes three variables (in blue) which are obtained at build-time; the rest are the variables that are copied from the “level.list” file.

```
   driverInfo Driver:  2001-03-09
buildModeInfo Build Mode:  Normal
 platformInfo This code was generated in  Windows NT Version 4.0
  productInfo IBM My Application
copyrightInfo (C) Copyright, IBM Corp.,1999,2001
  releaseInfo Release: 1.2.3.4
     dbmsInfo DBMS:    DB2 UDB 5.2 Fixpak 11
    patchInfo Patch:   0
```

* An include header file “what-info.h” and an include source file “what-info.c” are generated by the script “do-what” from the “master.list” file. Thus, the overall picture of the source code is the following:

The lines in blue are new or modified lines with respect to the original version.

```
myApp.c (main)
  |
  +---> Includes stdio and string
  |
  +---> Includes what-info.h and what-info.c
```

* The overall picture of the processing is the following:

The lines in blue are new or modified lines with respect to the original version.

```
do-build
  |
  +  Preparation step: set flags, deletes temp files
  |
  +  Copies the entries from "level.list" into "master.list"
  |
  +  Appends to "master.list" the rest of the variables
  |
  +  Invokes the make file with appropriate flags
                  |
                  +  Invokes do-what who reads the "master.list" to create:
                  |  what-info.h and what-info.c
                  |
                  +  Invokes C compiler
```

The implementation can be split into two stages:

a) Creation of the necessary files and updates to make files and source code files. This is done only once.

b) Update of the necessary files when performing a routine build, in which there is an update to the product.

Creation of the necessary files and updates to make files and source code files

The first step in the implementation of this method is the creation of the necessary files and the updates to the make files and the source code files. This is done only once.

* Create a file named “level.list” that has a set of variableName-variableValue pairs which specify the “static” searchable strings to be embedded in the executables. This file can be stored in a software configuration library. These are the strings that do not change very much.

* Create a script called “do-what.cmd” or “do-what.ksh” which will generate the include files:
    * what-info.h which is a header include file for the application.
    * what-info.c which is a source include file for the application.
    * what-info-level.c which is a source include file for the standalone level application.

* In the make file for the application, add an entry for the include file “what-info.h”. In that way, it is built from scratch. Also, add entries to invoke the “do-what” script.

* Create an include source code file called “what-info.c” which will make use of dummy statements with the serviceability strings. This is done to avoid the compiler’s optimizer from stripping the actual strings from the executable or shared library files: the idea is to both declare and use the strings, even though the usage is totally irrelevant (but will tell the optimizer to keep those strings!).

* Modify the source code for the executable file and add the include files “what-info.h” and “what-info.c”.

* Modify the script called “do-build” which invokes the “make” utility to build the product. This script can set some variables that might be included in the master.list file.

* Build the application and test the embedded serviceability strings.

The rest of this section provides actual samples that illustrate the points.

Create the “level.list” file with the “static” strings

The format is one row per entry; each entry is composed of a variable name, a space character and a variable value which is the rest of the row. For example:

```
  productInfo IBM My Application
copyrightInfo (C) Copyright, IBM Corp.,1999,2001
  releaseInfo Release: 1.2.3.4
     dbmsInfo DBMS:    DB2 UDB 5.2 Fixpak 11
    patchInfo Patch:   0
```

Create the script “do-what”

Create a script called “do-what.cmd” (in REXX) or “do-what.ksh” (in Korn shell) which will generate the include files “what-info” with all the desired serviceability strings. The sample script was developed to be self-contained, in the sense that if you want to add a new serviceability string, you do not need to modify this “do-what” script.

Korn shell script: do-what.ksh

```
#!/usr/bin/ksh
#
# NAME:  do-what.ksh levelListFile includeFileBaseName
#
# SAMPLE INVOCATION:
#
#   do-what.ksh master.list what-info
#
# PURPOSE:
#  To prepare the following include files with the information
#  of the serviceability strings specified in the levelListFile argument
#  (such as master.list).
#    * include header file, such as what-info.h
#    * include source file, such as what-info.c
#    * include source file for standalone level application,
#      such as what-info-level.c
#
# CUSTOMIZATION NOTES:
#  * See the sections labeled "CUSTOMIZATION REQUIRED".
#  * You may want to customize this script to suit your needs.
#    For example:
#    * Replace the value for the "keyword" variable to identify
#      your application.
#
#*********************************************************************/

# CUSTOMIZATION NEEDED.
# Set the keyword to be used during what and grep

keyword="myApp"

# Verify the input parameters

if [ "$#" -eq 2 ]
then
  levelListFile=$1
  includeFileBaseName=$2
  includeHeaderFile=‘echo $includeFileBaseName‘".h"
  includeSourceFile=‘echo $includeFileBaseName‘".c"
  includeSourceLevelFile=‘echo $includeFileBaseName‘"-level.c"
  print "do-what.ksh begin"
  print " -> file with level information: $levelListFile "
  print " -> include header file:         $includeHeaderFile"
  print " -> include source file:         $includeSourceFile"
  print " -> include source level file:   $includeSourceLevelFile"
else
  print "*** Error. Invalid number of arguments"
  print "Usage: do-what.ksh levelListFile includeFileBaseName"
  print ""
  exit 1
fi
```

```
#
# Generating the include header file
#

print ""
print "do-what.ksh: generating include header file: $includeHeaderFile"

rm $includeHeaderFile 2>/dev/null
print "/* Name:    $includeHeaderFile */"                          >>
$includeHeaderFile
print "/* Purpose: This file is generated during build time.    */" >>
$includeHeaderFile
print "/*  It has the 'build-time' serviceability information.  */" >>
$includeHeaderFile
print "/*  to be displayed by using the what utility.           */" >>
$includeHeaderFile
print ""                                                          >>
$includeHeaderFile
print "#ifndef _WHAT_INFO_H_"                                     >>
$includeHeaderFile
print "#define _WHAT_INFO_H_"                                     >>
$includeHeaderFile

while read line
do
   name=‘echo $line | cut -d' ' -f1‘   # Get first field
   value=‘echo $line | cut -d' ' -f2-‘ # Get everything after the first field
   stringValue=‘echo "char $name[] = \"@(#) $keyword $value\";"‘
   print "$stringValue"                                           >>
$includeHeaderFile
done < $levelListFile

print ""                                                          >>
$includeHeaderFile
print "#endif"                                                    >>
$includeHeaderFile
print ""                                                          >>
$includeHeaderFile
print "/* end of file */"                                         >>
$includeHeaderFile

#
# Generating the include generic source file
#

print ""
print "do-what.ksh: generating include generic source file:
$includeSourceFile"

rm $includeSourceFile 2>/dev/null

print "/* Name:    $includeSourceFile */"                           >>
$includeSourceFile
print "/* Purpose: This file is generated during build time.     */" >>
$includeSourceFile
print "/*   It has the 'build-time' serviceability information.  */" >>
$includeSourceFile
print "/*   to be displayed by using the what utility.           */" >>
$includeSourceFile
print "/* Note: It is necessary to declare and use the variables */" >>
$includeSourceFile
print "/*       in order to fool the compiler; otherwise, the    */" >>
$includeSourceFile
print "/*       compiler will remove them from the executable.   */" >>
$includeSourceFile
```

```
print ""                                                              >>
$includeSourceFile
print "int dummy=3;      /* Set to a number different than -7077 */" >>
$includeSourceFile
print "char dummyString[2000]; "                                      >>
$includeSourceFile
print ""                                                              >>
$includeSourceFile

while read line
do
   name=‘echo $line | cut -d’ ’ -f1‘    # Get first field

   stringValue="strcpy(dummyString,$name);"
   print "$stringValue"                                               >>
$includeSourceFile
done < $levelListFile

print ""                                                              >>
$includeSourceFile
print "if (dummy == -7077) {"                                         >>
$includeSourceFile
print "   printf(\"%s\",dummyString);"                                >>
$includeSourceFile
print "}"                                                             >>
$includeSourceFile

print ""                                                              >>
$includeSourceFile
print "/* end of file */"                                             >>
$includeSourceFile


#
# Generating the include level source file
#

print ""
print "do-what.ksh: generating include level source file:
$includeSourceLevelFile"

rm $includeSourceLevelFile 2>/dev/null

print "/* Name:    $includeSourceLevelFile */"                         >>
$includeSourceLevelFile
print "/* Purpose: This file is generated during build time.     */" >>
$includeSourceLevelFile
print "/*   It has the ’build-time’ serviceability information   */" >>
$includeSourceLevelFile
print "/*   to be displayed by the standalone level application. */" >>
$includeSourceLevelFile
print ""                                                              >>
$includeSourceLevelFile

while read line
do
   name=‘echo $line | cut -d’ ’ -f1‘    # Get first field

   stringValue="printf(\"%s\\\n\",$name);"
   print "$stringValue"                                               >>
$includeSourceLevelFile
done < $levelListFile
```

```
print ""                                                                >>
$includeSourceLevelFile
print "/* end of file */"                                              >>
$includeSourceLevelFile

print "do-what.ksh: done!"
print ""

exit 0

# end of file
```

Rexx command file: do-what.cmd

```
/* */
/*********************************************************************

NAME:      do-what.cmd levelFileList includeFileBaseName

SAMPLE INVOCATION:
  rexx do-what.cmd master.list what-info

PURPOSE:
 To prepare the following include files with the information
 of the serviceability strings specified in the levelListFile argument
 (such as level.list).
   * include header file, such as what-info.h
   * include source file, such as what-info.c
   * include source file for standalone level application,
     such as what-info-level.c

CUSTOMIZATION NOTES:
 * You may want to customize this script to suit your needs. There are
   some sections that are labeled "CUSTOMIZATION NEEDED". For example:
   * Replace the value for the "keyword" variable to identify
     your application.

*********************************************************************/

/* CUSTOMIZATION NEEDED.                                */
/* Set the keyword to be used during what and grep. */

keyword="myApp"

/* Verify the input parameters */

parse arg levelListFile includeFileBaseName .
usage = " usage: rexx do-what.cmd levelListFile includeFileBaseName"

/* Do the processing only if all variables are specified */

if (levelListFile="") | (includeFileBaseName="") then
do
  say "*** Error: parameters missing"
  say usage
  exit 1
end

say "do-what.cmd: begin"

/* Open input file */
rcFile = stream( levelListFile, C, 'open read' )
```

```
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " levelListFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/* Open output file: header file */
includeHeaderFile = includeFileBaseName || ".h"
rcFile = stream( includeHeaderFile, C, 'open write' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " includeHeaderFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/* Open output file: generic source file */
includeSourceFile = includeFileBaseName || ".c"
rcFile = stream( includeSourceFile, C, 'open write' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " includeSourceFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/* Open output file: level source file */
includeSourceLevelFile = includeFileBaseName || "-level.c"
rcFile = stream( includeSourceLevelFile, C, 'open write' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " includeSourceLevelFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/********* Create the output include header file ************/

say ""
say "do-what.cmd: generating include header file: " includeHeaderFile

outLine = '/* Name:  ' includeHeaderFile ' */'
temp = LINEOUT(includeHeaderFile, outLine);
outLine = '/* Purpose:  This file is generated during build time.  */'
temp = LINEOUT(includeHeaderFile, outLine);
outLine = '/*   It has the "build-time" serviceability information. */'
temp = LINEOUT(includeHeaderFile, outLine);
outLine = '/*   to be displayed by using the what utility.          */'
temp = LINEOUT(includeHeaderFile, outLine);
outLine = ''
temp = LINEOUT(includeHeaderFile, outLine);

outLine = '#ifndef _WHAT_INFO_H_'
temp = LINEOUT(includeHeaderFile, outLine);
outLine = '#define _WHAT_INFO_H_'
temp = LINEOUT(includeHeaderFile, outLine);
outLine = ''
temp = LINEOUT(includeHeaderFile, outLine);

DO UNTIL ( LINES( levelListFile ) = 0 )   /* Read until the end of file */
```

```
  pair = LINEIN( levelListFile )
  PARSE VAR pair variableName variableValue
  outLine="char " variableName"[] = ""@(#) " keyword variableValue " "";"
  temp = LINEOUT(includeHeaderFile, outLine)
END  /* do until */

outLine = '#endif'
temp = LINEOUT(includeHeaderFile, outLine);

outLine = ''
temp = LINEOUT(includeHeaderFile, outLine);

outLine = '/* end of file */'
temp = LINEOUT(includeHeaderFile, outLine);

rc = STREAM(includeHeaderFile, C, 'close');
rc = STREAM(levelListFile, C, 'close');

say "do-what.cmd: the include header file " includeHeaderFile " is now ready!"


/********* Create the output include generic source file ***********/

/* Open input file */
rcFile = stream( levelListFile, C, 'open read' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " levelListFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

say ""
say "do-what.cmd: generating include generic source file: " includeSourceFile

outLine = '/* Name:  ' includeSourceFile ' */'
temp = LINEOUT(includeSourceFile, outLine);
outLine = '/* Purpose:  This file is generated during build time.    */'
temp = LINEOUT(includeSourceFile, outLine);
outLine = '/*   It has the "build-time" serviceability information.  */'
temp = LINEOUT(includeSourceFile, outLine);
outLine = '/*   to be displayed by using the what utility.           */'
temp = LINEOUT(includeSourceFile, outLine);
outLine = "/* Note: It is necessary to declare and use the variables */"
temp = LINEOUT(includeSourceFile, outLine);
outLine = "/*       in order to fool the compiler; otherwise, the    */"
temp = LINEOUT(includeSourceFile, outLine);
outLine = "/*       compiler will remove them from the executable.   */"
temp = LINEOUT(includeSourceFile, outLine);
outLine = ''
temp = LINEOUT(includeSourceFile, outLine);

outLine = "int dummy=3;      /* Set to a number different than -7077 */"
temp = LINEOUT(includeSourceFile, outLine);
outLine = "char dummyString[2000]; "
temp = LINEOUT(includeSourceFile, outLine);
outLine = ""

DO UNTIL ( LINES( levelListFile ) = 0 )   /* Read until the end of file */
  pair = LINEIN( levelListFile )
  PARSE VAR pair variableName variableValue
  outLine="strcpy(dummyString," variableName "); "
  temp = LINEOUT(includeSourceFile, outLine)
END  /* do until */
```

```
outLine = ''
temp = LINEOUT(includeSourceFile, outLine);
outLine = "if (dummy == -7077) {"
temp = LINEOUT(includeSourceFile, outLine);
outLine = "   printf(""%s"",dummyString);"
temp = LINEOUT(includeSourceFile, outLine);
outLine = "}"
temp = LINEOUT(includeSourceFile, outLine);

outLine = ''
temp = LINEOUT(includeSourceFile, outLine);
outLine = '/* end of file */'
temp = LINEOUT(includeSourceFile, outLine);

rc = STREAM(includeSourceFile, C, 'close');
rc = STREAM(levelListFile, C, 'close');

say "do-what.cmd: the include source file " includeSourceFile " is now ready!"


/********* Create the output include level source file ************/

/* Open input file */
rcFile = stream( levelListFile, C, 'open read' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " levelListFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

say ""
say "do-what.cmd: generating include level source file: "
includeSourceLevelFile

outLine = '/* Name:  ' includeSourceLevelFile ' */'
temp = LINEOUT(includeSourceLevelFile, outLine);
outLine = '/* Purpose:  This file is generated during build time.    */'
temp = LINEOUT(includeSourceLevelFile, outLine);
outLine = '/*   It has the "build-time" serviceability information.  */'
temp = LINEOUT(includeSourceLevelFile, outLine);
outLine = '/*   to be displayed by the standalone level application */'
temp = LINEOUT(includeSourceLevelFile, outLine);
outLine = ''
temp = LINEOUT(includeSourceLevelFile, outLine);

DO UNTIL ( LINES( levelListFile ) = 0 )   /* Read until the end of file */
  pair = LINEIN( levelListFile )
  PARSE VAR pair variableName variableValue
  outLine="printf(""%s\n"","" variableName "); "
  temp = LINEOUT(includeSourceLevelFile, outLine)
END  /* do until */

outLine = ''
temp = LINEOUT(includeSourceLevelFile, outLine);
outLine = '/* end of file */'
temp = LINEOUT(includeSourceLevelFile, outLine);

rc = STREAM(includeSourceLevelFile, C, 'close');
rc = STREAM(levelListFile, C, 'close');

say "do-what.cmd: the include source file " includeSourceLevelFile " is now
ready!"
```

```
say "do-what.cmd: done!"

exit 0

/* end of file */
```

Add entry for what-info.h and the do-what script in the make file

In the make file for the application, add an entry for the include file “what-info.h”. In that way, it is built from scratch. Also, add entries to invoke the “do-what” script.

Notice that this makefile includes also “myAppLevel” which will be explained later.

Unix makefile: myApp-x.mak

The lines in blue are new or modified lines with respect to the original version.

```
# NAME: myApp-x.mak
#
# PURPOSE:
#   This is the makefile for building the myApp application in Unix.
#
# NOTES:
# * To build the application:
#      make -f myApp-x.mak myApp
# * The generation of the what-info* files is done by do-what.ksh
#

COMPILER=cc
CFLAGS=

all:  myApp myAppLevel

myApp: what-info.h main.o
      $(COMPILER) -o myApp myApp.o

myAppLevel: what-info.h myAppLevel.o
      $(COMPILER) -o myAppLevel myAppLevel.o

myApp.o: myApp.c
      $(COMPILER) $(FLAGS) -c myApp.c

mmyAppLevel.o: myAppLevel.c
      $(COMPILER) $(FLAGS) -c myAppLevel.c

what-info.h:
      ./do-what.ksh master.list what-info

# end of file
```

Windows makefile: myApp-w.mak

The lines in blue are new or modified lines with respect to the original version.

```
# NAME: myApp-w.mak
```

```
#
# PURPOSE:
#   This is the makefile for building the myApp application in Windows.
##
# NOTES:
# * To build the application using Windows 32-bit (VisualAge C++):
#      nmake -f myApp-w.mak myApp.exe
# * The generation of the what-info* files is done by do-what.cmd
#

COMPILER=icc

all:  myApp.exe myAppLevel.exe

myApp.exe: what-info.h myApp.obj
      $(COMPILER) /Fe myApp.exe myApp.obj

myAppLevel.exe: what-info.h myAppLevel.obj
      $(COMPILER) /Fe myAppLevel.exe myAppLevel.obj

myApp.obj: myApp.c
      $(COMPILER) $(FLAGS) /C+ myApp.c

myAppLevel.obj: myAppLevel.c
      $(COMPILER) $(FLAGS) /C+ myAppLevel.c

what-info.h:
      rexx .\do-what.cmd master.list what-info

# end of file
```

Detailed view of the generated source include file “what-info.c”

The script “do-what” will generate a source include file “what-info.c” which makes use of dummy statements with the serviceability strings. This is done to avoid having the compiler’s optimizer strip the actual strings from the executable or shared library files.

```
/* Name:   what-info.c  */
/* Purpose:  This file is generated during build time.    */
/*   It has the "build-time" serviceability information.  */
/*   to be displayed by using the what utility.           */
/* Note: It is necessary to declare and use the variables */
/*       in order to fool the compiler; otherwise, the    */
/*       compiler will remove them from the executable.   */

int dummy=3;      /* Set to a number different than -7077 */
char dummyString[2000];
strcpy(dummyString, driverInfo );
strcpy(dummyString, buildModeInfo );
strcpy(dummyString, platformInfo );
strcpy(dummyString, productInfo );
strcpy(dummyString, copyrightInfo );
strcpy(dummyString, releaseInfo );
strcpy(dummyString, dbmsInfo );
strcpy(dummyString, patchInfo );

if (dummy == -7077) {
   printf("%s",dummyString);
}

/* end of file */
```

Detailed view of the generated header include file “what-info.h”

The script “do-what” will generate a header include file “what-info.h”:

```
/* Name:   what-info.h  */
/* Purpose:  This file is generated during build time.  */
/*   It has the "build-time" serviceability information. */
/*   to be displayed by using the what utility.          */

#ifndef _WHAT_INFO_H_
#define _WHAT_INFO_H_

char  driverInfo[] = "@(#)  myApp Driver:  2001-03-09  ";
char  buildModeInfo[] = "@(#)  myApp Build Mode:  Normal  ";
char  platformInfo[] = "@(#)  myApp This code was generated in  Windows NT
Version 4.0    ";
char  productInfo[] = "@(#)  myApp IBM My Application  ";
char  copyrightInfo[] = "@(#)  myApp (C) Copyright, IBM Corp.,1999,2001  ";
char  releaseInfo[] = "@(#)  myApp Release: 1.2.3.4  ";
char  dbmsInfo[] = "@(#)  myApp DBMS:    DB2 UDB 5.2 Fixpak 11  ";
char  patchInfo[] = "@(#)  myApp Patch:   0  ";
#endif

/* end of file */
```

Detailed view of the generated source include file “what-info-level.c”

The script “do-what” will generate a source include file “what-info-level.c”, which is used in the standalone level application to actually print the serviceability strings:

```
/* Name:   what-info-level.c  */
/* Purpose:  This file is generated during build time.    */
/*   It has the "build-time" serviceability information.  */
/*   to be displayed by the standalone level application */

printf("%s\n", driverInfo );
printf("%s\n", buildModeInfo );
printf("%s\n", platformInfo );
printf("%s\n", productInfo );
printf("%s\n", copyrightInfo );
printf("%s\n", releaseInfo );
printf("%s\n", dbmsInfo );
printf("%s\n", patchInfo );

/* end of file */
```

Modify the source code for the executable file and add the include files

Modify the source code for the executable file and add the include files “what-info.h” and “what-info.c”. Notice the comment lines that describe important details when including these files.

The lines in blue are new or modified lines with respect to the original version.

```
/*
NAME:    myApp.c

PURPOSE:
   This file provides the main() for the sample application
   that will illustrate how to embed serviceability strings.

*/

#include <stdio.h>
#include <string.h>

/* Include the header file with the information for the 'what' command */

#include "what-info.h"

int main(void)
{

/* Include the source file with the information for the 'what' command */
/* It needs to be located after the last statement of the
   variable declaration section, because it will declare more variables
   and then use them. */

#include "what-info.c"
   printf("The myApp executable is running.\n");
   printf("It prints this message and then exits.\n");
   return 0;
}

/* end of file */
```

Update the script "do-build"

Update the script called "do-build.cmd" (in REXX) or "do-build.ksh" (in Korn shell) which invokes the "make" utility to build the product. This script can set some variables that might be included in the "master.list" file. This main build script needed substantial updates.

Korn shell script: do-build.ksh

```
#!/usr/bin/ksh
#
# NAME: do-build.ksh driverName [ debug | beta ]
#
# SAMPLE USAGES:
#   do-build.ksh  driver1
#   do-build.ksh  2001-03-01 debug
#
# PURPOSE:
#  This is the top level script to build the application "myApp".
#
# CUSTOMIZATION NOTES:
# * See the sections labeled "CUSTOMIZATION REQUIRED".
# * This example has 3 variables that need to be added to the master.list
#   file, because they change often and/or are set at build time.
#   + They need to be defined in Section "Definition" and
#   + They need to be added to the master.list in Section "Adding to List".
#   The variables are:
#    DRIVER_NAME      -> Identifier which might be related to the build date.
```

```
#                         This is passed as an input parameter to this script.
#    BUILD_MODE       -> Normal (default), or Debug or Beta
#    OPERATING_SYSTEM -> Using "uname -a" to find out the name.
#                         This is computed at build-time.
# * The file "level.list" has the serviceability strings that do not change
#   often. The contents of this file is copied into the "master.list".
# * The master.list file is a temporary file that contains all the
#   serviceability strings. This file is generated every single time
#   this build script is invoked and it is used inside the main make file
#   (myApp-x.mak) which in turn, calls do-what.ksh.
#
#****************************************************************

if [ $# -eq 0 ]
then
  print "Error: need to specify driver name."
  print "Usage: do-build.ksh driverName [ debug | beta ]"
  exit 1
fi

print "do-build.ksh: Starting a build of the myApp application"
print "date: ‘date‘"
print ""

# CUSTOMIZATION REQUIRED: Section "Definition"
# * Set up the variables that change frequently:
export BUILD_MODE="Normal"

# Find out the version and name of the operating system

export OPERATING_SYSTEM=‘uname -a‘
if [ "$OPERATING_SYSTEM" = "" ]
then
   print "*** Error: no output from uname"
   exit 1
fi

# CUSTOMIZATION REQUIRED:
# * You need to list the files to be removed before doing a build.

rm myApp             # This is the main application.
rm myAppLevel        # This is the standalone level application.
rm what-info.h       # This is the common include header file.
rm what-info.c       # This is the generic include source file.
rm what-info-level.c # This is the include source file for level application.
rm master.list       # This is a temporary file, with the complete list of
strings.
rm *.o


# CUSTOMIZATION REQUIRED:
# * You need to handle the appropriate processing of input arguments.
#   For example, the DEBUG and BETA options are set.

if [ $# -ge 1 ]
then
  typeset -u DRIVER_NAME=$1
  if [ $# -eq 2 ]
  then
    typeset -u OPTION=$2
    case "$OPTION" in
    BETA|beta)
      FLAGS=’-DBETA’
      export BUILD_MODE="BETA"
      ;;
```

```
    DEBUG|debug)
      FLAGS='-DDEBUG -g'
      export BUILD_MODE="DEBUG"
      ;;
    *)
      print "Unrecognized option: $OPTION"
      ;;
    esac
  fi
fi


# CUSTOMIZATION REQUIRED: Section "Adding to List"
# * It is necessary to create the master.list file with the variables
#   defined in this script. Follow the format below, which is also used
#   in the file with those serviceability strings that do not change often,
#   such as level.list. The format is a variableName (such as driverInfo)
#   followed by a value which can be several strings.
print "   driverInfo Driver: $DRIVER_NAME"      >>  master.list
print "buildModeInfo Build Mode: $BUILD_MODE"   >>  master.list
print " platformInfo This code was generated in $OPERATING_SYSTEM" >>
master.list


# CUSTOMIZATION REQUIRED:
# * If you use a name other than "level.list", then customize the following
#   statements.
levelListFile="level.list"
while read line

do

   print $line >> master.list
done < $levelListFile



# CUSTOMIZATION REQUIRED:
# * Invoke your make file

make -f myApp-x.mak all FLAGS="$FLAGS"

# end of file
```

Rexx command file: do-build.cmd

```
/* */
/*********************************************************************

NAME: do-build.cmd driverName [ debug | beta ]

SAMPLE INVOCATION:
  rexx do-build.cmd 2001-03-09 debug

PURPOSE:
 This is the top level script to build the application "myApp".

NOTES:
 * A temporary file named "do-ver.out" is used.

CUSTOMIZATION NOTES:
 * See the sections labeled "CUSTOMIZATION REQUIRED".
```

```
 * This example has 3 variables that need to be added to the master.list
   file, because they change often and/or are set at build time.
   + They need to be defined in Section "Definition" and
   + They need to be added to the master.list in Section "Adding to List".
   The variables are:
    DRIVER_NAME      -> Identifier which might be related to the build date.
    BUILD_MODE       -> Normal (default), or Debug or Beta
    OPERATING_SYSTEM -> Using "uname -a" to find out the name.
 * The file "level.list" has the serviceability strings that do not change
   often. The contents of this file is copied into the "master.list".
 * The master.list file is a temporary file that contains all the
   serviceability strings. This file is generated every single time
   this build script is invoked and it is used inside the main make file
   (myApp-w.mak) which in turn, calls do-what.cmd.

*********************************************************************/

say "do-build.ksh: Starting a build of the myApp application"
say "date: "
"@date /t"
say ""

/* Verify the input parameters */

parse arg driverName option
usage = "do-build.cmd driverName [ debug | beta ]"

/* CUSTOMIZATION REQUIRED: Section "Definition"
   Set up the variables that change frequently:
*/
if (driverName = "") then
do
  say "*** Error: need to specify the driver name."
  say usage
  exit 1
end
else
do
  DRIVER_NAME=driverName
end

BUILD_MODE='Normal'

/* Issue the "ver" command and store the output in a temporary file named
   "do-ver.out".
   The variable 'OPERATING_SYSTEM' will have the version of the operating
system.
*/
verout = "do-ver.out"

"@ver >" verout
 do while lines(verout)
    parse value linein(verout) with OPERATING_SYSTEM
 end
 call lineout verout
 "@del" verout

/* Remove files */

/* CUSTOMIZATION REQUIRED:
   You need to list the files to be removed before doing a build.
*/

'@del what-info.h'       /* This is the common include header file. */
'@del what-info.c'       /* This is the generic include source file. */
```

```
'@del what-info-level.c' /* This is the include source file for level applica-
tion. */
'@del master.list'       /* This is a temporary file, with the complete list
of strings. */
'@del *.obj'
'@del *.exe'

/* CUSTOMIZATION REQUIRED:
   You need to handle the appropriate processing of input arguments.
   For example, the DEBUG and BETA options are set.
*/

/* Do the processing for the option */

if (option = "") then
do
  FLAGS=""
end
else
do
  if (option = "debug") then
  do
     FLAGS="/Ti+ /DDEBUG"
     BUILD_MODE='DEBUG'
  end
  else
  do
     if (option = "beta") then
     do
        FLAGS="/DBETA"
        BUILD_MODE='BETA'
     end
     else
     do
       say "*** Error: wrong option. If specified, it must be: debug or beta"
       say usage
       exit 1
     end
  end
end

/* CUSTOMIZATION REQUIRED:
   Specify the permanent file "level.list" with the variables that do not
   change often. This is the input file.
   Specify the temporary file "master.list" that will have all the
   strings. This is the output file.
*/

/* Open input file */
inputFile='level.list'
rcFile = stream( inputFile, C, 'open read' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " inputFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/* Open output file */
outputFile='master.list'
rcFile = stream( outputFile, C, 'open write' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " outputFile
```

```
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/* CUSTOMIZATION REQUIRED: Section "Adding to List"
   It is necessary to create the master.list file with the variables
  defined in this script. Follow the format below, which is also used
  in the file with those serviceability strings that do not change often,
  such as level.list. The format is a variableName (such as driverInfo)
  followed by a value which can be several strings.
*/

outLine="   driverInfo Driver: " DRIVER_NAME
temp = LINEOUT(outputFile, outLine);
outLine="buildModeInfo Build Mode: " BUILD_MODE
temp = LINEOUT(outputFile, outLine);
outLine=" platformInfo This code was generated in " OPERATING_SYSTEM
temp = LINEOUT(outputFile, outLine);

DO UNTIL ( LINES( inputFile ) = 0 )   /* Read until the end of file */
  outLine = LINEIN( inputFile )
  temp = LINEOUT(outputFile,outLine)
END  /* do until */

rc = STREAM(outputFile, C, 'close');
rc = STREAM(inputFile, C, 'close');

/* Invoke the make file */

say 'nmake -f myApp-w.mak all FLAGS='FLAGS
    'nmake -f myApp-w.mak all FLAGS="'FLAGS'"'
/* end of file */
```

Build the application and test the embedded serviceability strings.

Now that all the components are ready, you can proceed to build the application and test the embedded serviceability strings.

Unix: Execute the build script and test the serviceability strings (normal mode):

The following command will generate the executable myApp:

```
$ do-build.ksh 2001-03-09
```

The output messages of the executable myApp are:

```
$ myApp
The myApp executable is running.
It prints this message and then exits.
```

Use the 'what' utility to see the embedded serviceability strings:

```
$ what myApp
myApp:
 61      1.11  src/bos/usr/ccs/lib/libc/__threads_init.c, libcthrd, bos43
```

```
K, 9823A 43K 6/12/98 12:37:06
 myApp Driver: 2001-03-09
 myApp Build Mode: Normal
 myApp This code was generated in AIX tcaix08 3 4 006081014C00
 myApp IBM My Application
 myApp (C) Copyright, IBM Corp.,1999,2001
 myApp Release: 1.2.3.4
 myApp DBMS: DB2 UDB 5.2 Fixpak 11
 myApp Patch: 0
```

Notice that there was an extra line in the output that had nothing to do with the application. Thus, it is possible to issue the following to filter only those strings for “myApp”:

```
$ what myApp | grep myApp
myApp:
 myApp Driver: 2001-03-09
 myApp Build Mode: Normal
 myApp This code was generated in AIX tcaix08 3 4 006081014C00
 myApp IBM My Application
 myApp (C) Copyright, IBM Corp.,1999,2001
 myApp Release: 1.2.3.4
 myApp DBMS: DB2 UDB 5.2 Fixpak 11
 myApp Patch: 0
```

Windows: Execute the build script and test the serviceability strings (normal):

```
c:\> rexx do-build.cmd 2001-03-09
```

The output messages of the executable myApp are:

```
c:\>myApp
The myApp executable is running.
It prints this message and then exits.

c:\> what myApp.exe
MyApp.exe:
 myApp Driver: 2001-03-09
 myApp Build Mode: Normal
 myApp Code generated in Platform: Windows NT Version 4.0
 MyApp IBM My Application
 myApp (C) Copyright, IBM Corp.,1999,2001
 myApp Release: 1.2.3.4
 myApp DBMS: DB2 UDB 5.2 Fixpak 11
 myApp Patch: 0
```

Unix: Execute the build script and test the serviceability strings (beta):

```
$ do-build.ksh 2001-03-09 beta

$ what myApp | grep myApp
myApp:
 myApp Driver: 2001-03-09
 myApp Build Mode: Beta
 myApp This code was generated in AIX tcaix08 3 4 006081014C00
 myApp IBM My Application
```

```
 myApp (C) Copyright, IBM Corp.,1999,2001
 myApp Release: 1.2.3.4
 myApp DBMS: DB2 UDB 5.2 Fixpak 11
 myApp Patch: 0
```

Windows: Execute the build script and test the serviceability strings (beta):

```
c:\> rexx do-build.cmd 2001-03-09 beta

c:\> what myApp.exe
MyApp.exe:
 myApp Driver: 2001-03-09
 myApp Build Mode: Beta
 myApp Code generated in Platform: Windows NT Version 4.0
 MyApp IBM My Application
 myApp (C) Copyright, IBM Corp.,1999,2001
 myApp Release: 1.2.3.4
 myApp DBMS: DB2 UDB 5.2 Fixpak 11
 myApp Patch: 0
```

Update of the necessary files when performing a build for an update to the product

The second step in this method is simply the maintenance of the necessary files when performing a build for an update to the product.

* If you need to add another “static” string, the only change is to add it to the “level.list” file, which is a file that should be under library control.

* If you need to add another kinds of serviceability string, then you need to add it in the script “do-build”, and follow the customization notes to properly include all the necessary statements.

* Build the application and test the embedded serviceability strings.

How to implement the standalone tool in C language

This section describes how to prepare a standalone serviceability tool that displays the desired serviceability strings. This application could be built using make files in multiple platforms using multiple DBMSs; specifically, the cases for Windows NT and AIX are thoroughly explained.

This standalone serviceability tool needs to be built and delivered with the normal application.

The files mentioned in this section are available from the "c" subdirectory from the zip file referred to in the section "How to get the files mentioned in this document ".

This section is a continuation of the previous section “How to embed the serviceability strings in C language”, in which scripts, makefiles and included files are described in detail. This current chapter covers only the details on the source file for the standalone executable “myAppLevel” which displays the desired serviceability strings. By the way, the makefiles from the previous chapter already handle the compilation and linking of this standalone tool.

Create the main source code file “myAppLevel.c”

Create a source code file called “myAppLevel.c” which will make use of dummy statements with the serviceability strings and will display them.

Notice that you can add extra variables that are not defined in what-info.h and what-info.c, such as “dummyPatch” in order to display the patch (also called hot-fix or temporary fix) for the application.

```
/*
NAME:    myAppLevel.c

PURPOSE:
   This file provides the main() for the sample standalone
   tool for an application that will illustrate how to embed
   serviceability strings.

NOTES:
   To compile:
     Unix:      cc myAppLevel.c -o myAppLevel
     Windows:  icc myAppLevel.c /Fe myAppLevel
*/

#include <stdio.h>
#include <string.h>

/* Include the header file with the information for the 'what' command */
#include "what-info.h"

int main(void)
{
/* Include the source file with the information for the 'what' command */
/* It needs to be located after the last statement of the variable
   declaration section, because it will declare more variables
```

```
   and then use them. */
#include "what-info-level.c"

   return 0;
}

/* end of file */
```

Unix: Execute the build script and test the serviceability strings (normal):

```
$ do-build.ksh 2001-03-09 beta

$ myAppLevel
@(#) myApp Driver: 2001-03-09
@(#) myApp Build Mode: BETA
@(#) myApp This code was generated in AIX tcaix08 3 4 006081014C00
@(#) myApp IBM My Application
@(#) myApp (C) Copyright, IBM Corp.,1999,2001
@(#) myApp Release: 1.2.3.4
@(#) myApp DBMS: DB2 UDB 5.2 Fixpak 11
@(#) myApp Patch: 0
```

Windows: Execute the build script and test the serviceability strings (beta):

```
c:\> rexx do-build.cmd 2001-03-09

c:\> myAppLevel.exe
@(#)  myApp Driver:  2001-03-09
@(#)  myApp Build Mode:  Normal
@(#)  myApp This code was generated in  Windows NT Version 4.0
@(#)  myApp IBM My Application
@(#)  myApp (C) Copyright, IBM Corp.,1999,2001
@(#)  myApp Release: 1.2.3.4
@(#)  myApp DBMS:    DB2 UDB 5.2 Fixpak 11
@(#)  myApp Patch:   0
```

How to embed the serviceability strings in Java

This section describes how to embed the serviceability strings in the executable file for a sample application called “myapp” which is developed in the Java language. This application could be built using make files on multiple platforms using multiple DBMSs. In what follows, the process is thoroughly documented for AIX and Windows NT.

The files in this section are available from the “java” subdirectory of the expanded zip file referred to in the section “How to get the files mentioned in this document “.

The structure of this chapter is as follows:

- Part 1: Original structure of the files.
  A sample application called “myapp” is shown in its original form, which does not use the methodology explained in this document. The source code is very simple because we are not concerned with other aspects of the development of an application. Also, the original Korn shell scripts and Windows command batch files are shown. This provides a baseline that will give the reader a point of reference.
- Part 2: Modifications to include the serviceability strings
  The source code will be modified to contain the searchable strings mentioned in this document. The objective is to show the reader, the relatively small “delta” effort that is needed in order to implement the methodology.

Part 1: Original structure of the files

Let’s start with the description of the simple application myapp which is written in Java and which does not contain the serviceability strings. The idea is to provide a baseline to help understand the modifications explained later on.

* The overall picture of the source code is the following:

```
myApp.java (main)
```

* The overall picture of the processing is the following:

```
do-build
  |
  +  Preparation step: set flags, delete temp files
  |
  +  Invoke the make file
                  |
                  +  Invoke Java compiler
```

Notice that this is a very small and simple program.

Original source code for myapp.java

```
/*
NAME:    myapp.java

PURPOSE:
   This file provides the main() for the sample application
   that will illustrate how to embed serviceability strings.

SAMPLE INVOCATION:
   java myapp.myapp
*/


package myapp;

public class myapp
{
  //VARIABLES
    public static final boolean DEBUG = false;
    public static final String debugInfo = "*** This is DEBUG code ***";
    public static final boolean BETA  = false;
    public static final String betaInfo = "*** This is BETA code ***";

  public static void main(String[] args)
  {
    System.out.println("The base myapp executable is running.");
    System.out.println("It prints this message and then exits.");

  }//end main()
}//end myapp
```

Original source code for the make files

Unix makefile: myApp-x.mak

```
# NAME: myapp-x.mak
#
# PURPOSE:
#   This is the makefile for building the myapp application in UNIX.
#
# NOTES:
# * To build the application:
#      make -f myapp-x.mak myapp
#

COMPILER=javac



all:  clean myapp

myapp: myapp/myapp.class

myapp/myapp.class: myapp/myapp.java
      $(COMPILER) myapp/myapp.java

clean:
      -cd myapp; rm *.class; cd ..
# end of file
```

Windows makefile: myApp-w.mak

```
# NAME: myApp-w.mak
#
# PURPOSE:
#   This is the makefile for building the myApp application in Windows.
#
# NOTES:
# * To build the application using Windows 32-bit (VisualAge C++):
#      nmake -f myApp-w.mak myApp.exe
#

COMPILER=icc

all:  myApp.exe

myApp.exe: myApp.obj
      $(COMPILER) /Fe myApp.exe myApp.obj

myApp.obj: myApp.c
      $(COMPILER) $(FLAGS) /C+ myApp.c

# end of file
```

Original source code for the scripts to invoke the make file

Korn shell script: do-build.ksh

```
#!/usr/bin/ksh
#
# NAME: do-build.ksh driverName [ debug | beta ]
#
# SAMPLE USAGES:
#   do-build.ksh  driver1
#   do-build.ksh  2001-03-01 debug
#
# PURPOSE:
#  This is the top level script to build the application "myapp".
#

#*****************************************************************



# * You need to handle the appropriate processing of input arguments.
#   For example, the DEBUG and BETA options are set.

if [ $# -ge 1 ]
then
  typeset -u DRIVER_NAME=$1
  if [ $# -eq 2 ]
  then
    typeset -u OPTION=$2
    case "$OPTION" in
    BETA|beta)
       print "**********ERROR**********"
       print "Set the BETA variable in the java source code for myapp "
       print "to 'true' and then call this script without the beta parameter."
```

```
       print ""
       print "./do-build.ksh driverName"
       exit 1

      ;;
    DEBUG|debug)
     print "**********ERROR**********"
     print "Set the DEBUG variable in the java source code for myapp "
     print "to 'true' and then call this script without the debug  parameter:"
     print ""
     print "./do-build.ksh driverName"
     exit 1

      ;;
    *)
      print "Unrecognized option: $OPTION"
      ;;
    esac
  fi
fi

# * Invoke your make file

make -f myapp-x.mak all FLAGS="$FLAGS"

# end of file
```

Original Source Code for the build Scripts

Rexx command file: do-build.cmd

```
/* */
/*********************************************************************

NAME: do-build.cmd [ debug | beta ]

SAMPLE INVOCATION:
  rexx do-build.cmd 2001-03-09 debug

PURPOSE:
 This is the top level script to build the application "myapp".

*********************************************************************/


/* Verify the input parameters */

parse arg driverName option
usage = "do-build.cmd [ debug | beta ]"

/* Do the processing for the option */

if (option = "") then
do
  FLAGS=""
end
else
do
  if (option = "debug") then
  do
     say "**********ERROR**********"
```

```
     say "Set the DEBUG variable in the java source code for myapp "
     say "to 'true' and then call this script without the debug  parameter:"
     say ""
     say "rexx do-build.cmd driverName"
     exit 1

  end
  else
  do
     if (option = "beta") then
     do
       say "**********ERROR**********"
       say "Set the BETA variable in the java source code for myapp "
       say "to 'true' and then call this script without the beta parameter."
       say ""
       say "rexx do-build.cmd driverName"
       exit 1
     end
     else
     do
       say "*** Error: wrong option. If specified, it must be: debug or beta"
       say usage
       exit 1
     end
  end
end



/* Section:  Invoke the make file */

say 'nmake -f myapp-w.mak all FLAGS='FLAGS''
'nmake -f myapp-w.mak all FLAGS="'FLAGS'"'
/* end of file */
```

Files needed:

java\base\do-build.cmd
java\base\do-build.ksh
java\base\myapp-w.mak
java\base\myapp-x.mak
java\base\myapp\myapp.java

Building the base myapp application:

The steps that follow are for both Windows (INTEL) and AIX (UNIX) platforms since the java code is built on and runs on both platforms in essentially the same way.

1) First create the files above or unzip the trlevel.zip file described in the section "How to get the Files Mentioned in this Document.".

2) Add the location of the myapp directory to the CLASSPATH variable

```
Intel:  set CLASSPATH=%CLASSPATH%;<theLocationOfThemyappDirectory>
 UNIX:  export CLASSPATH=$CLASSPATH:<theLocationOfThemyappDirectory>
```

For example,

```
set CLASSPATH=%CLASSPATH%;D:\rick\papers\trlevel\java\base
```

*NOTE: it is important to have your CLASSPATH set properly, including having the current working directory (.) in the CLASSPATH. Here is an example:

```
CLASSPATH=.;c:\jdk1.1.8\lib\classes.zip;c:\jdk1.1.8\lib;D:\rick\papers\trlevel
\java\base
```

2) cd into the java\base directory (java/base on UNIX) and build the myapp and myapplevel executables by running the do-build script. NOTE: UNIX users must run a CRLF tool on all files to remove carriage return characters (^M).

```
Syntax:  do-build.cmd driverName [ debug | beta ]

Intel:  rexx do-build.cmd
 UNIX:  chmod 755 do-build.ksh
       ./do-build.ksh
```

3) Test the myapp application by running it.

java myapp.myapp

The myapp application only prints out a message that it was running and then exits:

```
The base myapp executable is running.
It prints this message and then exits.
```

In real life the myapp application would be your main application to which you wish to add the serviceability strings.

4) Test the standard "what" command against the myapp executable.

```
what myapp\myapp.class
```

It does not display any information because there are no serviceability strings in the product.

myapp\myapp.class:

Part 2: Modifications to include the serviceability strings

There are 2 types of serviceability strings, from the point of view of their creation times. All these strings will eventually be placed in the java class whatInfo.

- Strings that seldom change, such as the copyright information, the product name, the company name. For this reason, in this document these strings are referred as “static strings”. They are defined in the “level.list” file and handled inside the script “do-what”. level.list is a permanent file that once created can be checked-out from, modified and then checked-in to a library system when one of these variables needs to be changed (not a daily event). We felt that it was too cumbersome to include those variables that may change daily, because we did not want to have the overhead of checking out and checking in the file every day.
- Strings that change frequently or that can be obtained from the environment or require additional processing. It is not feasible to add them into the level.list file because it would cause a lot of overhead with the software configuration system. Instead, these variables are handled by the main build script “do-build” and then added to a temporary “master.list” file, which in turn is handled by the script “do-what”. A sub-classification of these variables is shown below:
    - Strings that change frequently and which are specified by the user, usually when a build activity is performed, such as the driver/level. These strings are referred to as “build-time strings”. They are passed as parameters or as environment variables to the script “do-build”.
    - Strings that can be obtained from the environment, such as the version of the operating system used during the build. These strings are referred to as “environmental strings”.
    - Strings that do not fit neatly into the above categories, such as when building beta code or debug code. These strings are referred as “miscellaneous strings”.

For the java version of myapplevel there is also another type of string --- the build mode which is embedded within the myapp application itself. These are hard-coded strings that reside within the main application and which represent either DEBUG or BETA builds. These are the only strings that are not included in the master.list. The myapplevel application will retrieve the information about the build mode directly from the myapp application.

The main modifications to include the serviceability strings are:

* A permanent file, named “level.list”, that has a set of variable-value pairs which specify the “static” searchable strings to be embedded in the executables. This file can be stored in a software configuration library. These are the strings that do not change very much. The format is one row per entry; each entry is composed of a variable name, a space and a variable value which is the rest of the row. For example:

```
  productInfo IBM My Application
copyrightInfo (C) Copyright, IBM Corp.,1999,2001
  releaseInfo Release: 1.2.3.4
     dbmsInfo DBMS:    DB2 UDB 5.2 Fixpak 11
    patchInfo Patch:   0
```

* A temporary file, named “master.list”, which will be recreated during each build and which will have the strings from the “level.list” file and will have additional strings (this is done inside do-build). Eventually this master.list will be handled by the script “do-what” to generate the whatInfo class which is described next. For example, the master.list file includes three variables

(in blue) which are obtained at build-time; the rest are the variables that are copied from the "level.list" file.

```
   driverInfo Driver:  2001-03-09
buildModeInfo Build Mode:  Normal
 platformInfo This code was generated in  Windows NT Version 4.0
  productInfo IBM My Application
copyrightInfo (C) Copyright, IBM Corp.,1999,2001
  releaseInfo Release: 1.2.3.4
     dbmsInfo DBMS:    DB2 UDB 5.2 Fixpak 11
    patchInfo Patch:   0
```

* The whatInfo class is generated by the script "do-what" from the "master.list" file. Thus, the overall picture of the source code is the following:.

```
myApp.java (main)
```

* The overall picture of the processing is the following:

The lines in blue are new or modified lines with respect to the original version.

```
do--build
  |
  +  Preparation step: set flags, deletes temp files
  |
  +  Copies "static" variables from "level.list" into the "master.list"
  |
  +  Prepends the "dynamic" variables to the "master.list"
  |
  +  Invokes the make file
                  |
                  +  Invokes do-what which reads the "master.list" to create:
                  |  whatInfo.java
                  |
                  +  Invokes Java compiler
```

The implementation can be split into two stages:

a) Creation of the necessary files and updates to make files and source code files. This is done only once.

b) Update of the necessary files when performing a routine build, in which there is an update to the product.

These two steps are discussed in detail next.

Creation of the necessary files and updates to make files and source code files

The first step in the implementation of this method is the creation of the necessary files and the updates to the make files and the source code files. This is done only once.

* Create a file named “level.list” that has a set of variableName-variableValue pairs which specify the “static” searchable strings to be embedded in the executables. This file can be stored in a software configuration library. These are the strings that do not change very much.

* Create a script called “do-what.cmd” or “do-what.ksh” which will generate the whatInfo class:
    * whatInfo.java is a class for the myapplevel application.

* In the make file for the application, add an entry for the class file “whatInfo.java”. In that way, it is built from scratch. Also, add entries to invoke the “do-what” script.

* Modify the script called “do-build” which invokes the “make” utility to build the product. This script can set some variables that might be included in the master.list file.

* Build the myapp and myapplevel applications and test the embedded serviceability strings.

The rest of this section provides actual samples that illustrate the points.

Create the “level.list” file with the “static” strings

The format is one row per entry; each entry is composed of a variable name, a space character and a variable value which is the rest of the row. For example:

```
  productInfo IBM My Application
copyrightInfo (C) Copyright, IBM Corp.,1999,2001
  releaseInfo Release: 1.2.3.4
     dbmsInfo DBMS:    DB2 UDB 5.2 Fixpak 11
    patchInfo Patch:   0
```

Create the script “do-what”

Create a script called “do-what.cmd” (in REXX) or “do-what.ksh” (in Korn shell) which will generate the class file “whatInfo” with all the desired serviceability strings. The sample script was developed to be self-contained, in the sense that if you want to add a new serviceability string, you do not need to modify this “do-what” script.

Korn shell script: do-what.ksh

```
#!/usr/bin/ksh
#
# NAME:  do-what.ksh levelListFile includeFileBaseName
#
# SAMPLE INVOCATION:
```

```
#
#   do-what.ksh master.list what-info
#
# PURPOSE:
#
# PURPOSE:
# To generate the java class whatInfo which will be instantiated
# by both the myapp and myapplevel applications.  The "report"
# method will be used to display all the serviceability strings.

# CUSTOMIZATION NOTES:
#  * See the sections labeled "CUSTOMIZATION REQUIRED".
#  * You may want to customize this script to suit your needs. For example:
#    * Replace the value for the "keyword" variable to identify
#      your application.
#
#*****************************************************************/

# CUSTOMIZATION NEEDED.
# Set the keyword to be used during what and grep

keyword="myapp"

# Verify the input parameters

if [ "$#" -eq 2 ]
then
  levelListFile=$1
  includeFileBaseName=$2
  infoFile=‘echo $includeFileBaseName‘".java"
  print "do-what.ksh begin"
  print " -> file with level information: $levelListFile "
  print " -> info file:         $infoFile"
else
  print "*** Error. Invalid number of arguments"
  print "Usage: do-what.ksh levelListFile includeFileBaseName"
  print ""
  exit 1
fi

#
# Generating the java info file
#

print ""
print "do-what.ksh: generating info file: $infoFile"

rm $infoFile 2>/dev/null
print "/* Name:    $infoFile */"                              >> $infoFile
print "/* Purpose: This file is generated during build time.    */" >>
$infoFile
print "/*  It has the ’build-time’ serviceability information.  */" >>
$infoFile
print "/*  to be displayed by using the myapplevel utility.           */" >>
$infoFile
print ""                                                              >>
$infoFile
print "package $keyword;"                                            >>
$infoFile
print ""                                                              >>
$infoFile
print "public class $includeFileBaseName"                            >>
$infoFile
print "{"                                                            >>
$infoFile
```

```
print "  //This class is generated by do-what.ksh or do-what.cmd.  Do not
modify!" >> $infoFile
print "  //VARIABLES"                                                  >>
$infoFile

while read line
do
   name=‘echo $line | cut -d’ ’ -f1‘    # Get first field
   value=‘echo $line | cut -d’ ’ -f2-‘  # Get everything after the first field

   stringValue=‘echo "  public String $name = \"@(#) $keyword $value\";"‘
   print "$stringValue"                                                  >>
$infoFile
done < $levelListFile

print "  //METHODS"                                                 >> $infoFile
print "    public void report()"                                   >> $infoFile
print "    {"                                                      >> $infoFile
print ""                                                           >> $infoFile
while read line
do
   name=‘echo $line | cut -d’ ’ -f1‘    # Get first field
   value=‘echo $line | cut -d’ ’ -f2-‘  # Get everything after the first field

   stringValue=‘echo "  System.out.println(this.$name);"‘
   print "$stringValue"                                                  >>
$infoFile
done < $levelListFile
print "    }//end of report()"                                     >> $infoFile
print "}//end of whatInfo"                                        >> $infoFile
print ""                                                           >> $infoFile
print "/* end of file */"                                         >> $infoFile



print "do-what.ksh: done!"
print ""

exit 0

# end of file
```

Rexx command file: do-what.cmd

```
/* */
/*********************************************************************


NAME:      do-what.cmd levelFileList includeFileBaseName

SAMPLE INVOCATION:
  rexx do-what.cmd master.list whatInfo

PURPOSE:
To generate the java class whatInfo which will be instantiated
by both the myapp and myapplevel applications.  The "report"
method will be used to display all the serviceability strings.

CUSTOMIZATION NOTES:
 * You may want to customize this script to suit your needs. There are
   some sections that are labeled "CUSTOMIZATION NEEDED". For example:
   * Replace the value for the "keyword" variable to identify
     your application.
```

```
*********************************************************************/

/* CUSTOMIZATION NEEDED.                                */
/* Set the keyword to be used during what and grep. */

keyword="myapp"

/* Verify the input parameters */

parse arg levelListFile includeFileBaseName .
usage = " usage: rexx do-what.cmd levelListFile includeFileBaseName"

/* Do the processing only if all variables are specified */

if (levelListFile="") | (includeFileBaseName="") then
do
  say "*** Error: parameters missing"
  say usage
  exit 1
end

say "do-what.cmd: begin"

/* Open input file */
rcFile = stream( levelListFile, C, 'open read' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " levelListFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/* Open output file: */
infoFile = includeFileBaseName || ".java"
rcFile = stream( infoFile, C, 'open write' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " infoFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end


/********* Create the info java file ************/

say ""
say "do-what.cmd: generating info file: " infoFile

outLine = '// Name:  ' infoFile
temp = LINEOUT(infoFile, outLine);
outLine = '// Purpose:  This file is generated during build time.   '
temp = LINEOUT(infoFile, outLine);
outLine = '//   It has the "build-time" serviceability information. '
temp = LINEOUT(infoFile, outLine);
outLine = '//   to be displayed by using the what utility.          '
temp = LINEOUT(infoFile, outLine);
outLine = ''
temp = LINEOUT(infoFile, outLine);

outLine = 'package ' keyword ';'
temp = LINEOUT(infoFile, outLine);
outLine = ''
```

```
temp = LINEOUT(infoFile, outLine);

outLine = 'public class ' includeFileBaseName
temp = LINEOUT(infoFile, outLine);

outLine = '{'
temp = LINEOUT(infoFile, outLine);

outLine = '  //This class is generated by do-what.ksh or do-what.cmd.  Do not
modify!'
temp = LINEOUT(infoFile, outLine);

outLine = '  //VARIABLES'
temp = LINEOUT(infoFile, outLine);

DO UNTIL ( LINES( levelListFile ) = 0 )   /* Read until the end of file */
  pair = LINEIN( levelListFile )
  PARSE VAR pair variableName variableValue
  outLine = "  public String " variableName" = ""@(#) " keyword variableValue
" "";"
  temp = LINEOUT(infoFile, outLine)
END  /* do until */

outLine = ''
temp = LINEOUT(infoFile, outLine);

outLine = '  //METHODS'
temp = LINEOUT(infoFile, outLine);

outLine = '    public void report()'
temp = LINEOUT(infoFile, outLine);

outLine = '    {'
temp = LINEOUT(infoFile, outLine);

outLine = ''
temp = LINEOUT(infoFile, outLine);

rc = STREAM(levelListFile, C, 'close');
/* Open input file */
rcFile = stream( levelListFile, C, 'open read' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " levelListFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

DO UNTIL ( LINES( levelListFile ) = 0 )   /* Read until the end of file */
  pair = LINEIN( levelListFile )
  PARSE VAR pair variableName variableValue
  outLine = "     System.out.println(this."variableName");"
  temp = LINEOUT(infoFile, outLine)
END  /* do until */


outLine = '    }//end of report()'
temp = LINEOUT(infoFile, outLine);

outLine = '    }//end of' includeFileBaseName
temp = LINEOUT(infoFile, outLine);

outLine = ''
temp = LINEOUT(infoFile, outLine);
```

```
outLine = '/* end of file */'
temp = LINEOUT(infoFile, outLine);

rc = STREAM(infoFile, C, 'close');
rc = STREAM(levelListFile, C, 'close');

say "do-what.cmd: the info class file " infoFile " is now ready!"

exit 0

/* end of file */
```

Add entry for whatInfo.java and the do-what script in the make file

In the make file for the (myapp) application, add an entry for the class file "whatInfo.java". In that way, it is built from scratch. Also, add entries to invoke the "do-what" script.

Notice that this makefile also includes "myapplevel" which will be explained later.

Unix makefile: myApp-x.mak

```
# NAME: myapp-x.mak
#
# PURPOSE:
#   This is the makefile for building the myapp application in UNIX.
#
# NOTES:
# * To build the application:
#      make -f myapp-x.mak myapp
#

COMPILER=javac



all:  clean myapp myapplevel

myapp: myapp/whatInfo.class myapp/myapp.java
      $(COMPILER) myapp/myapp.java

myapplevel: myapp/whatInfo.class myapp/myapplevel.java
      $(COMPILER) myapp/myapplevel.java


myapp/whatInfo.class:
      cd myapp; ./do-what.ksh ../master.list whatInfo; cd ..
      $(COMPILER) myapp/whatInfo.java

clean:
      -cd myapp; rm *.class; rm whatInfo.java; cd ..
# end of file
```

Windows makefile: myApp-w.mak

The lines in blue are new or modified lines with respect to the original version.

```
# NAME: myapp-w.mak
```

```
#
# PURPOSE:
#   This is the makefile for building the myapp application in Windows.
#
# NOTES:
# * To build the application:
#      nmake -f myapp-w.mak myapp
#

COMPILER=javac



all:  clean myapp myapplevel

myapp: myapp\whatInfo.class myapp\myapp.java
      $(COMPILER) myapp\myapp.java

myapplevel: myapp\whatInfo.class myapp\myapplevel.java
      $(COMPILER) myapp\myapplevel.java


myapp\whatInfo.class:
      cd myapp
      rexx .\do-what.cmd ..\master.list whatInfo
      cd ..
      $(COMPILER) myapp\whatInfo.java

clean:
      cd myapp
      -del *.class
      -del whatInfo.java
      cd ..
# end of file
```

Detailed view of the generated class file “whatInfo.java”

The script “do-what” will generate a header include file “whatInfo.java”:

```
// Name:   whatInfo.java
// Purpose:  This file is generated during build time.
//   It has the "build-time" serviceability information.
//   to be displayed by using the what utility.

package  myapp ;

public class  whatInfo
{
  //This class is generated by do-what.ksh or do-what.cmd.  Do not modify!
  //VARIABLES
  public String  driverInfo = "@(#)  myapp Driver:  2001-03-09  ";
  public String  buildModeInfo = "@(#)  myapp Build Mode:  Normal  ";
  public String  platformInfo = "@(#)  myapp This code was generated in
Windows NT Version 4.0    ";
  public String  productInfo = "@(#)  myapp IBM My Application  ";
  public String  copyrightInfo = "@(#)  myapp (C) Copyright, IBM
Corp.,1999,2001  ";
  public String  releaseInfo = "@(#)  myapp Release: 1.2.3.4  ";
  public String  dbmsInfo = "@(#)  myapp DBMS:    DB2 UDB 5.2 Fixpak 11  ";
  public String  patchInfo = "@(#)  myapp Patch:   0  ";
  public String  developerInfo = "@(#)  myapp Developers:  Rick Russell and
Angel Rivera  ";
```

```
  //METHODS
    public void report()
    {

      System.out.println(this.driverInfo);
      System.out.println(this.buildModeInfo);
      System.out.println(this.platformInfo);
      System.out.println(this.productInfo);
      System.out.println(this.copyrightInfo);
      System.out.println(this.releaseInfo);
      System.out.println(this.dbmsInfo);
      System.out.println(this.patchInfo);
    }//end of report()
    }//end of whatInfo

/* end of file */
```

Source code for the executable file myapp

It is not necessary to modify the source code for the myapp application other than to add the word myapp and a few leading spaces to the front of DEBUG and BETA strings so that appear lined up with the rest of the output from the myapplevel application.

```
/*
NAME:    myapp.java

PURPOSE:
   This file provides the main() for the sample application
   that will illustrate how to embed serviceability strings.

SAMPLE INVOCATION:
   java myapp.myapp
*/


package myapp;

public class myapp
{
  //VARIABLES
    public static final boolean DEBUG = false;
    public static final String debugInfo = "      myapp *** This is DEBUG code
***";
    public static final boolean BETA  = false;
    public static final String betaInfo = "      myapp *** This is BETA code
***";

  public static void main(String[] args)
  {
    System.out.println("The myapp executable is running.");
    System.out.println("It prints this message and then exits.");

  }//end main()
}//end myapp
```

Update the script “do-build”

Update the script called “do-build.cmd” (in REXX) or “do-build.ksh” (in Korn shell) which invokes the “make” utility to build the product. This script can set some variables that might be included in the “master.list” file. This main build script needed substantial updates.

Korn shell script: do-build.ksh

```
#!/usr/bin/ksh
#
# NAME: do-build.ksh driverName [ debug | beta ]
#
# SAMPLE USAGES:
#   do-build.ksh  driver1
#   do-build.ksh  2001-03-01 debug
#
# PURPOSE:
#  This is the top level script to build the application "myapp".
#
# NOTES:
# * A temporary file named "do-ver.out" is used.
#
# CUSTOMIZATION NOTES:
# * See the sections labeled "CUSTOMIZATION REQUIRED".
# * This example has variables that need to be added to the master.list
#   file, because they change often and/or are set at build time.
#   - They need to be defined in Section "Definition of Dynamic Variables"
#   - They need to be added to the master.list in Section "Create
master.list".
#   The variables are:
#    DRIVER_NAME      -> Identifier which might be related to the build date.
#    OPERATING_SYSTEM -> Use "ver" or "uname -a" to find out the OS level.
# * The file "level.list" has the serviceability strings that do not change
#   often. The contents of this file are copied into the "master.list".
# * The master.list file is a temporary file that contains all the
#   serviceability strings. master.list is generated every single time
#   this build script is invoked and it is used inside the main make file
#   (myapp-x.mak) which in turn, calls do-what.cmd.
#
#
#*****************************************************************

if [ $# -eq 0 ]
then
  print "Error: need to specify driver name."
  print "Usage: do-build.ksh driverName [ debug | beta ]"
  exit 1
fi

print "do-build.ksh: Starting a build of the myapp application"
print "date: ‘date‘"
print ""


# Find out the version and name of the operating system

export OPERATING_SYSTEM=‘uname -a‘
if [ "$OPERATING_SYSTEM" = "" ]
then
   print "*** Error: no output from uname"
   exit 1
fi
```

```
# CUSTOMIZATION REQUIRED:
# * You need to handle the appropriate processing of input arguments.
#   For example, the DEBUG and BETA options are set.

if [ $# -ge 1 ]
then
  typeset -u DRIVER_NAME=$1
  if [ $# -eq 2 ]
  then
    typeset -u OPTION=$2
    case "$OPTION" in
    BETA|beta)
       print "**********ERROR**********"
       print "Set the BETA variable in the java source code for myapp "
       print "to 'true' and then call this script without the beta parameter."
       print ""
       print "./do-build.ksh driverName"
       exit 1

      ;;
    DEBUG|debug)
     print "**********ERROR**********"
     print "Set the DEBUG variable in the java source code for myapp "
     print "to 'true' and then call this script without the debug  parameter:"
     print ""
     print "./do-build.ksh driverName"
     exit 1

      ;;
    *)
      print "Unrecognized option: $OPTION"
      ;;
    esac
  fi
fi

# CUSTOMIZATION REQUIRED:
# You need to remove the old master.list before doing a build.
# NOTE:  not really customization.

rm master.list       # This is a temporary file, with the complete list of
strings.

# CUSTOMIZATION REQUIRED:
#   You need to add variables that do not change often to the file
#   "level.list".  This is the input file and is permanent.
#   "master.list" is a temporary file that will have all the
#   serviceability strings. This is the output file.


# CUSTOMIZATION REQUIRED: Section:  "Create master.list"
# * It is necessary to create the master.list file with the variables
#   defined in this script. Follow the format below, which is also used
#   in the file with those serviceability strings that do not change often,
#   such as level.list. The format is a variableName (such as driverInfo)
#   followed by a value which can be several strings.
print "   driverInfo Driver: $DRIVER_NAME"      >>  master.list
print " platformInfo This code was generated in $OPERATING_SYSTEM" >>
master.list


# CUSTOMIZATION REQUIRED:
# * If you use a name other than "level.list", then customize the following
#   statement.
```

```
levelListFile="level.list"
while read line

do

   print $line >> master.list
done < $levelListFile



# CUSTOMIZATION REQUIRED:
# * Invoke your make file

make -f myapp-x.mak all FLAGS="$FLAGS"

# end of file
```

Rexx command file: do-build.cmd

```
/* */
/*********************************************************************

NAME: do-build.cmd driverName [ debug | beta ]

SAMPLE INVOCATION:
  rexx do-build.cmd 2001-03-09 debug

PURPOSE:
 This is the top level script to build the application "myapp".

NOTES:
 * A temporary file named "do-ver.out" is used.

CUSTOMIZATION NOTES:
 * See the sections labeled "CUSTOMIZATION REQUIRED".
 * This example has variables that need to be added to the master.list
   file, because they change often and/or are set at build time.
   - They need to be defined in Section "Definition of Dynamic Variables"
   - They need to be added to the master.list in Section "Create
master.list".
   The variables are:
    DRIVER_NAME      -> Identifier which might be related to the build date.
    OPERATING_SYSTEM -> Use "ver" or "uname -a" to find out the OS level.
 * The file "level.list" has the serviceability strings that do not change
   often. The contents of this file are copied into the "master.list".
 * The master.list file is a temporary file that contains all the
   serviceability strings. master.list is generated every single time
   this build script is invoked and it is used inside the main make file
   (myapp-w.mak) which in turn, calls do-what.cmd.

*********************************************************************/

say "do-build.cmd: Starting a build of the myapp application"
say "date: "
"@date /t"
say ""

/* Verify the input parameters */

parse arg driverName option
usage = "do-build.cmd driverName [ debug | beta ]"
```

```
/* Section:  Definition of Dynamic Variables */

/* CUSTOMIZATION REQUIRED:
   Set up the variables that change frequently:
*/
if (driverName = "") then
do
  say "*** Error: need to specify the driver name."
  say usage
  exit 1
end
else
do
  DRIVER_NAME=driverName
end


/* Issue the "ver" command and store the output in a temporary file named
   "do-ver.out".
   The variable 'OPERATING_SYSTEM' will have the version of the operating
system.
*/
verout = "do-ver.out"

"@ver >" verout
 do while lines(verout)
    parse value linein(verout) with OPERATING_SYSTEM
 end
 call lineout verout
 "@del" verout

/* CUSTOMIZATION REQUIRED:
   You need to handle the appropriate processing of input arguments.
   For example, the DEBUG and BETA options may need to be set.
*/

/* Do the processing for the option */

if (option = "") then
do
  FLAGS=""
end
else
do
  if (option = "debug") then
  do
     say "**********ERROR**********"
     say "Set the DEBUG variable in the java source code for myapp "
     say "to 'true' and then call this script without the debug  parameter:"
     say ""
     say "rexx do-build.cmd driverName"
     exit 1

  end
  else
  do
     if (option = "beta") then
     do
       say "**********ERROR**********"
       say "Set the BETA variable in the java source code for myapp "
       say "to 'true' and then call this script without the beta parameter."
       say ""
       say "rexx do-build.cmd driverName"
       exit 1
     end
```

```
     else
     do
       say "*** Error: wrong option. If specified, it must be: debug or beta"
       say usage
       exit 1
     end
  end
end

/* Section:  Remove files */

/* CUSTOMIZATION REQUIRED:
   You need to remove the old master.list before doing a build.
NOTE:  not really customization.
*/

'@del master.list'            /* generated file with the complete list of
strings. */

/* CUSTOMIZATION REQUIRED:
   You need to add variables that do not change often to the file
   "level.list".  This is the input file and is permanent.
   "master.list" is a temporary file that will have all the
   serviceability strings. This is the output file.
*/

/* Open input file */
/* CUSTOMIZATION REQUIRED:
 * If you use a name other than "level.list", then customize the following
   statement. */
inputFile='level.list'
rcFile = stream( inputFile, C, 'open read' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " inputFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end

/* Open output file */
outputFile='master.list'
rcFile = stream( outputFile, C, 'open write' )
if ( rcFile \= "READY:" ) then
do
   say "*** Error while trying to open file: " outputFile
   say "    The return code is: " rcFile
   say "    Exiting now."
   exit 1
end


/* Section:  Create  master.list" */

/* CUSTOMIZATION REQUIRED:
   It is necessary to create the master.list file with the variables
  defined in this script. Follow the format below, which is also used
  in the file level.list which contains serviceability strings that do
  not change often.  The format is a variableName (such as driverInfo)
  followed by a value which can be several strings.
*/

outLine="   driverInfo Driver: " DRIVER_NAME
temp = LINEOUT(outputFile, outLine);
```

```
outLine=" platformInfo This code was generated in " OPERATING SYSTEM
temp = LINEOUT(outputFile, outLine);

DO UNTIL ( LINES( inputFile ) = 0 )   /* Read until the end of file */
  outLine = LINEIN( inputFile )
  temp = LINEOUT(outputFile,outLine)
END  /* do until */

rc = STREAM(outputFile, C, 'close');
rc = STREAM(inputFile, C, 'close');

/* Section:  Invoke the make file */

say 'nmake -f myapp-w.mak all FLAGS='FLAGS''
'nmake -f myapp-w.mak all FLAGS="'FLAGS'"'
/* end of file */
```

Create the main source code file "myapplevel.java" for the standalone tool myapplevel

Create a source code file called "myapplevel.java". This is a standalone application which will ship with the main application myapp and which will display the serviceability strings.

```
package myapp;
/*
NAME:    myapplevel.java

PURPOSE:
   This file provides the main() for the application
   myapplevel which retrieves serviceability strings
   from the application myapp.

SAMPLE INVOCATION:
   java myapp.myapplevel


*/
import myapp.whatInfo;

public class myapplevel
{

  //VARIABLES
    whatInfo wi          = null;

  //METHODS
  public myapplevel()
  {
    this.wi  = new whatInfo();
  }//end myapplevel

  public static void main(String[] args)
  {
    myapplevel mal = new myapplevel();

    if(myapp.DEBUG == true)
      System.out.println(myapp.debugInfo);
    if(myapp.BETA  == true)
      System.out.println(myapp.betaInfo);

    //Print out the serviceability strings.
    mal.wi.report();
```

```
  }//end main()
}//end myapplevel
```

Build the application and test the embedded serviceability strings.

Now that all the components are ready, you can proceed to build the application and test the embedded serviceability strings.

Files Needed:
java\do-build.cmd
java\do-build.ksh
java\level.list
java\myapp-w.mak
java\myapp-x.mak
java\myapp\do-what.cmd
java\myapp\do-what.ksh
java\myapp\myapp.java
java\myapp\myapplevel.java

Building myapp and myapplevel and testing the serviceability strings:

The steps that follow are for both Windows (INTEL) and AIX (UNIX) platforms since the java code is built on and runs on both platforms in essentially the same way.

1) First create the files above or unzip the trlevel.zip file described in the section "How to get the Files Mentioned in this Document.".

2) Add the location of the myapp directory to the CLASSPATH variable

```
Intel:  set CLASSPATH=%CLASSPATH%;<theLocationOfThemyappDirectory>
UNIX:  export CLASSPATH=$CLASSPATH:<theLocationOfThemyappDirectory>
```

For example,

```
set CLASSPATH=%CLASSPATH%;D:\rick\papers\trlevel\java
```

*NOTE: it is important to have your CLASSPATH set properly, including having the current working directory (.) in the CLASSPATH. Here is an example:

```
CLASSPATH=.;c:\jdk1.1.8\lib\classes.zip;c:\jdk1.1.8\lib;D:\rick\papers\trlevel
\java
```

2) cd into the java directory and build the myapp and myapplevel executables by running the do-build script. NOTE: UNIX users must run a CRLF tool on all files to remove carriage return characters (^M).

```
Syntax:  do-build.cmd driverName [ debug | beta ]

Intel:  rexx do-build.cmd 2001-03-09
 UNIX:  chmod 755 do-build.ksh
        chmod 755 myapp/do-what.ksh
        ./do-build.ksh 2001-03-09
```

3) Test the myapp application by running it.

```
java myapp.myapp
```

The myapp application only prints out a message that it was running and then exits:

```
The myapp executable is running.
It prints this message and then exits.
```

In real life the myapp application would be your main application to which you wish to add the serviceability strings.

4) Test the standard "what" command against the myapp executable.

```
what myapp\myapp.class
```

which results in

```
myapp\myapp.class:
```

Since myapp is object-oriented and most serviceability strings are not stored within the myapp class, "what" will not show the strings. If you wished to see the serviceability strings using the “what” utility, you could issue the “what” command against the whatInfo class:

```
Intel:  what myapp\whatInfo.class
 UNIX:  what myapp/whatinfo.class
```

which would produce

```
myapp\whatInfo.class:
    myapp (C) Copyright, IBM Corp.,1999,2001
    myapp DBMS:    DB2 UDB 5.2 Fixpak 11
    myapp Driver:  2001-03-09
    myapp IBM My Application
    myapp Patch:   0
    myapp Release: 1.2.3.4
    myapp This code was generated in  Windows NT Version 4.0
```

5) Test the myapplevel application

```
java myapp.myapplevel
```

You should see

```
@(#)  myapp Driver:  2001-03-09
@(#)  myapp Build Mode:  Normal
```

```
@(#)  myapp This code was generated in  Windows NT Version 4.0
@(#)  myapp IBM My Application
@(#)  myapp (C) Copyright, IBM Corp.,1999,2001
@(#)  myapp Release: 1.2.3.4
@(#)  myapp DBMS:    DB2 UDB 5.2 Fixpak 11
@(#)  myapp Patch:   0
```

Doing DEBUG or BETA builds:

do-build.cmd (do-build.ksh) will allow you to specify either debug or beta builds. However, since Java does not have conditional compilation, the debug, beta, and normal builds are specified within the code of myapp.java by setting the variables DEBUG or BETA to either "true" or "false". Thus to set up a debug build you would set DEBUG=true within myapp.java and rebuild the code using the do-build script. The do-build script will print out a message instructing you to do this.

```
Intel:  rexx do-build.cmd 2001-03-09 debug
 UNIX:  ./do-build.ksh 2001-03-09 debug
```

results this message on Windows (Intel)

```
**********ERROR**********
Set the DEBUG variable in the java source code for myapp
to 'true' and then call this script without the debug  parameter:

rexx do-build.cmd driverName
```

and this on AIX (UNIX)

```
**********ERROR**********
Set the DEBUG variable in the java source code for myapp
to 'true' and then call this script without the debug  parameter:

./do-build.cmd driverName
```

To perform a debug build, you need to open the file myapp.java and change the following line

```
public static final boolean DEBUG = false;
```

to look like this

```
public static final boolean DEBUG = true;
```

Now, run the do-what script again without adding the debug argument:

```
Intel:  rexx do-build.cmd 2001-03-09
 UNIX:  ./do-build.ksh 2001-03-09
```

Once that is done myapplevel will read the DEBUG string directly out of the java myapp.class file and "java myapp.myapplevel" results in

```
      myapp *** This is DEBUG code ***
@(#)  myapp Driver:  2001-03-09
@(#)  myapp Build Mode:  Normal
@(#)  myapp This code was generated in  Windows NT Version 4.0
@(#)  myapp IBM My Application
@(#)  myapp (C) Copyright, IBM Corp.,1999,2001
@(#)  myapp Release: 1.2.3.4
@(#)  myapp DBMS:    DB2 UDB 5.2 Fixpak 11
@(#)  myapp Patch:   0
```

Beta builds follow the same process.

Update of the necessary files when performing a build for an update to the product

The first part of this method to add serviceability strings to your application was to create all the necessary files and rebuild the myapp and myapplevel applications. The second step in this method is simply the maintenance of the necessary files when performing a build for an update to the product. The actions required depend on whether the serviceability string you wish to add is dynamic or static.

* If you need to add another “static” string, the only change is to add it to the “level.list” file, which is a file that should be under library control.

* If you need to add another kind of serviceability string, then you need to add it in the script “do-build”, and follow the customization notes to properly include all the necessary statements.

* Build the application and test the embedded serviceability strings. The two types of strings are discussed in the next two points.

Adding static serviceability strings to level.list

You can add new static serviceability strings to the level.list file. For example, if you wished to add a serviceability string with the name of the program developers, you would add it to level.list since (in this example anyway) the names of the developers is unlikely to change. The new entry in level.list would appear as:

developerInfo Developers: Rick Russell and Angel Rivera

After rebuilding with the do-build script, "java myapp.myapplevel" reports:

```
@(#)  myapp Driver:  2001-03-20
@(#)  myapp This code was generated in  Windows NT Version 4.0
@(#)  myapp IBM My Application
@(#)  myapp (C) Copyright, IBM Corp.,1999,2001
@(#)  myapp Release: 1.2.3.4
@(#)  myapp DBMS:    DB2 UDB 5.2 Fixpak 11
```

```
@(#)  myapp Patch:   0
@(#)  myapp Developers:  Rick Russell and Angel Rivera
```

Adding dynamic serviceability strings to do-build.cmd:

You can add serviceability strings for dynamic information to the do-build script. This implies that you need to add the additional functionality to do-build.cmd or do-build.ksh to determine what the string should be. For instance, you might want to record the time the executables were built. This could be accomplished by retrieving the date and time from the operating system. Possibly there are environment variables that are used in the build of your product. One such environment variable might be MYAPP_RELEASE which you set to the current version of the release being built. You could retrieve this information from MYAPP_RELEASE within the do-build script and then set a serviceability string equal to that value. In this way the Release string that used to come from level.list has become dynamic and would now be calculated. There is no overriding rule that determines whether a particular serviceability string has to be dynamic (do-build) or static (level.list).

How to get the files mentioned in this document

Notes for Windows users:
- The “grep” utility is included in the Microsoft Windows Services for Unix (SFU). For more details visit the URL: http://www.microsoft.com/windows2000/sfu/
- The scripts are written in REXX.

All the tools used here are available via the public Internet. The tools might be updated in the future. The tools are zipped into a single file called **trlevel.zip** and it can be downloaded as follows:

FTP site for TeamConnection

You can download the code from our external FTP site for TeamConnection:

ftp://ftp.software.ibm.com/ps/products/teamconnection/papers/trlevel.zip

Obtaining a tool that fix Carriage Return and Line Feed problems

In case that you have problems with the carriage return and line feed when trying to view or compile the source code files, you can use the fixcrlf tool provided in the following ftp site:

ftp://ftp.software.ibm.com/ps/products/teamconnection/tools/

Obtaining Info-ZIP

The VisualAge TeamConnection team uses the Info-Zip **zip** and **unzip** tools to package compressed files (in which the files to be packaged are compressed first).

The main advantages of Info-ZIP are:
- Compatibility: these tools are compatible with other ZIP programs.
- Portability: they are available in ALL the platforms that are supported by VisualAge TeamConnection.
- Cross-platform: A zip file prepared in Unix can be unzipped in the correct format in Windows NT and vice versa.

Info-ZIP's software is free and can be obtained from:
ftp://ftp.uu.net:/pub/archiving/zip/

Because of the general value of these tools, it is recommended that you add the unzip and zip tools in a directory in the PATH that is accessible to all the users for the machine, such as C:\WINNT.

How to unzip files
- To only view the contents of the zip file (without actually unpackaging and uncompressing the files) do: **unzip -l trlevel.zip**

- To unpackage and uncompress the zip file do: **unzip trlevel.zip**

Copyrights, Trademarks and Service marks

The following terms used in this technical report, are trademarks or service marks of the indicated companies:

TRADEMARK, REGISTERED TRADEMARK, OR SERVICE MARK	COMPANY
IBM, VisualAge, TeamConnection, DB2 Universal Database, CMVC	IBM Corporation
Unix	Unix System Laboratories, Inc.
Windows, Windows NT	Microsoft Corporation
Intel	Intel Corporation
Info-ZIP	Info-ZIP Group
Java, Solaris	Sun Microsystems, Corp.
HP-UX	Hewlett-Packward Corp.

***** END OF DOCUMENT *****